ON LINUX PERFORMANCE OF CPU-BOUND PROCESSES IN THE PRESENCE OF NETWORK I/O

BY

ABDULRAHMAN MOHAMMAD AL-MANA

A Thesis Presented to the

DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER SCIENCE

JUNE, 2009

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN 31261, SUADI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis written by **ABDULRAHMAN MOHAMMAD AL-MANA** under the direction of his thesis advisor and approved by his thesis committee, has been presented to and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN COMPUTER SCIENCE**.

Thesis Committee

Dr. Khaled Salah (Thesis Advisor)

Dr. Farag Azzedin

Dr. Salahadin Adem Mohammed

Department Chairman
Dr. Kanaan Faisal

Dean of Graduate Studies
Dr. Salam A. Zummo

8/7/09
Date

Dedication

To my dear mother

Hessah

And my beloved wife

Bayan

for their endless encouragement, prayers and sacrifices.

ACKNOWLEDGMENT

First and foremost, all praise is due to Allah the Almighty without Whose aid nothing could be possible. It was definitely impossible to complete this work without His assistance. My efforts could have been abortive without His Facilitations. I praise and thank Him abundantly for the bountiful blessings which He has bestowed upon me.

Upon the completion of this thesis, I wish to express my deepest gratitude for my thesis advisor, Dr. Khaled Salah, for his indispensible advice and tremendous, long-term support. I sincerely appreciate his invaluable guidance in the shaping of this work, sharing both his time and expertise. The assistance, advice, encouragement and invaluable support he has given me throughout the course of this work and on several other occasions are much more than my simple thanks and gratitude.

Thanks are extended to the members of the thesis committee, Dr. Farag Azzedin and Dr. Salahadin Adem Mohammed for their help, support, and contributions. I am truthfully indebted to them for their generous sharing of knowledge, valuable scientific comments, and professional advice.

Finally, I wish to express my deepest gratitude to my family members, especially my mother and wife, for being patient with me and offering words of encouragements to spur my spirit at moments of depression. They patiently endured with me the hard times and the critical periods I went through while completing the thesis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

THESIS ABSTRACT

NAME: Abdulrahman Mohammad Al-Mana

TITLE: On Linux Performance of CPU-Bound Processes in the Presence of Network I/O

MAJOR FIELD: Computer Science

DATE OF DEGREE: June 2009

In this research work, we demonstrate that Linux can starve CPU-bound processes in the presence of network I/O-bound processes. Surprisingly, the starvation of CPU-bound processes can be only encountered at a particular range of traffic rates being received by the network processes. Lower or higher traffic rates do not exhibit starvation. We demonstrate experimentally the existence of such a starvation problem in the presence of a number of different network applications, and under different system settings and network configurations. We show that the problem exists for the two versions of Linux Scheduler, namely the 2.6 O(1) scheduler and the more recent 2.6 Completely Fair Scheduler (CFS). We instrumented and profiled the Linux kernel to investigate the underlying root cause of such surprising behavior and starvation. To alleviate the starvation problem, we proposed and implemented a solution for both 2.6 O(1) and CFS schedulers. We demonstrate that our solution is highly effective in improving the performance of CPU-bound processes in both uni-processing (UP) and symmetric multiprocessing (SMP) Linux systems. Our experimental results show that the proposed solution has no negative impact on the performance of network processes.

خلاصة الرسالة

الإســـــم : عبدالرحمن محمد عبدالرحمن المانع

عنوان الرسالة : دراسة في أداء العمليات المرتبطة بوحدة المعالجة المركزية في ظل تواجد نشاط شبكي في نظام تشغيل لينوكس

التخصص : علوم الحاسب الآلي

تاريخ التخرج : يونيو 2009

الدراسة المقدمة في هذا البحث تسلط الضوء على إمكانية حدوث تراجع شديد في أداء العمليات المرتبطة بوحدة المعالجة المركزية للحاسب في نظام تشغيل لينوكس، في ظل تواجد عمليات مرتبطة بمعالجة بيانات الشبكة. المدهش أن التراجع الشديد في أداء العمليات المرتبطة بوحدة المعالجة المركزية لا يحدث إلا في مجال محدد من حركة سير بيانات الشبكة، بينما لا يلاحظ هذا التراجع خارج هذا ا لنطاق من حركة سير بيانات الشبكة. قمنا في هذا البحث بإثبات وجود هذه الظاهرة عمليا في ظل وجود تطبيقات شبكية مختلفة، و باستخدام أشكال متعددة من إعدادات نظام التشغيل و الشبكة. بالإضافة لما سبق،لقد تم إثبات وجود هذه الظاهرة في إصدارين مختلفين من نظام الجدولة في لينوكس، و هما: نظام الجدولة المعروف بـــ O(1) 2.6 ونظام الجدولة الأحدث، المعروف بـــنظام الجدولة ذي العدالة الكاملة (CFS). قمنا بتعديل برمجة نظام لينوكس للوقوف أكثر على المشكلة و تحليل السبب الرئيسي وراء هذه الظاهرة الغريبة. لعلاج هذه الظاهرة، تم اقتراح وتطبيق حل ممكن لكل من النسختين من نظام جدولة لينوكس. وضحنا عمليا كيف أن الحل المقترح فعال جدا في تحسين أداء العمليات المرتبطة بوحدة المعالجة المركزية للحاسب في بيئة نظام تشغيل لينوكس أحادية و متعددة المعالجة. نتائج التجارب تثبت أيضا عدم وجود أي تأثير سلبي للحل المقترح على أداء عمليات الشبكة.

CHAPTER 1

INTRODUCTION

Linux is an open-source operating system initially developed by Linus Torvalds in 1991 [LOV05]. In fact, Linux has gained, and is still gaining, a great popularity from both individuals and business sectors because of its simplicity, flexibility and robustness. Several versions of Linux are continuously developed and enhanced by both individual groups and professional companies. Several components of Linux are usually modified from version to version to fix their existing problems and enhance their performance.

One of the most important parts of Linux kernel is the process scheduler. The process scheduler is the component of the kernel responsible for selecting which process to run next on the CPU. Thus, the scheduler is of crucial importance, since it is executed so often in one side, and it is responsible for dividing the CPU resources among different competing processes in the other side.

The following subsection gives a brief explanation of the concepts and terminologies used with operating system scheduler. Next, a brief introduction to Linux scheduling algorithm (particularly, the Linux 2.6 O(1) scheduler) is presented.

1.1 Process Scheduler

One of the most significant components of any operating system is the processor scheduler, the component of the operating system kernel that is used to choose which process should be executed next, and thus controls and distributes the resources of the processor among several contending processes in the system [LOV05]. Obviously, the scheduler is executed much more often than most other components of the operating system. Therefore, the selection of a suitable scheduling algorithm and an efficient implantation of such an algorithm are very critical to the operating system reliability and performance.

In multitasking operating systems (operating systems that interleaves the execution of several concurrent processes, such as Linux), the scheduler plays the dominant role in the implementation of concurrent processing. By deciding which process to run next, the scheduler tries to optimize the utilization of the system resources, giving the illusion that multiple processes are running in the same time. There are two types of multitasking in operating system: cooperative and preemptive. In cooperative multitasking operating systems, a running process is only stopped from execution on a voluntarily basis. On the other hand, running processes in preemptive multitasking operating systems are forcibly stopped from execution, so-called "preempted", when the scheduler decides that another process should be executed. Linux is an example of a preempted multitasking operating system [LOV05].

From a scheduling prospective, processes are usually classified as CPU bound or I/O bound. CPU bound processes are those processes which spends most of their

lifetime either executing or waiting to be executed. Those processes are not blocked because of I/O or memory requests. Rather, they are preempted from execution after they have totally used their share from the CPU time, usually called timeslice. Thus, they wait to get a new timeslice. On the other hand, I/O bound processes are processes that usually initiate many I/O requests. As the CPU speed is much more than the I/O drivers, those processes get blocked until their I/O requests are completed. Therefore, those processes spend most of their lifetime waiting for their I/O requests. An optimal goal of the scheduler is to achieve the maximum CPU utilization by evenly permuting CPU bound and I/O bound processes [LOV05].

In many operating systems, scheduling involves assigning different priorities to different kinds of processes. Those priorities are usually numerical values that the scheduler uses to differentiate between processes according to their need for the processor time. In some operating systems, the priorities are dynamic, and they are adjusted according to the behavior of the process. For example, in Linux, the priorities of processes that spend most of their time waiting for I/O requests are increased, to give them more chance of execution upon completion of their I/O requests. On the other hand, the priorities of processes that consume all of their CPU timeslice and get preempted are decreased, to prevent them from monopolizing the CPU [LOV05].

Any scheduling algorithm has to satisfy basic constraints to be effective and usable. To begin with, the scheduling algorithm must be fair in dealing with different processes. In other words, given two identical processes, the scheduling algorithm has to give equal treatment to both of them. In addition, the scheduling algorithm must prevent starvation of processes. More clearly, there must be a limited amount of time that a

ready process will wait before it gets executed. Moreover, the scheduling algorithm has to be efficient in consuming processing power. Furthermore, the scheduling overhead imposed by executing the scheduling algorithm has to be minimal. Additional features related to multiuser/multitasking operating systems are good response time for interactive processes, and intelligent treatment of CPU bound and I/O bound processes [SIL03].

1.2 Scheduling in Linux 2.6

One of the major changes in the kernel of Linux 2.6, released to the public in December 2003, was an entirely new scheduling algorithm. This scheduling algorithm was advertised for its O(1) time complexity, and thus it was widely called: Linux 2.6 O(1) Scheduler. This scheduler gained a wide acceptance among the Linux community through the years, though it was recently replaced in kernel 2.6.23 with a new scheduling algorithm, namely, the Completely Fair Scheduler (CFS). Linux 2.6 O(1) scheduler was designed to achieve the following new features:

- A bound of O(1) on the time needed to choose the next process to run.
- Quick response to interactive processes under high system load.
- An acceptable level of prevention of both starving and hogging.
- Scalability and task affinity under symmetric multiprocessing environment.
- Improved performance when having a small number of processes [TOR07], [LOV05].

In general, the main data structure used by the Linux scheduler is the runqueue, which contains the tasks that are ready to run. The runqueue is divided internally into two arrays: the active array and the expired array. The active array contains all processes that are ready to run. On the other hand, the expired array contains all processes that are temporarily disabled as they have used their entire timeslices. When the active array becomes empty, the expired array becomes again active [TOR07], [LOV05].

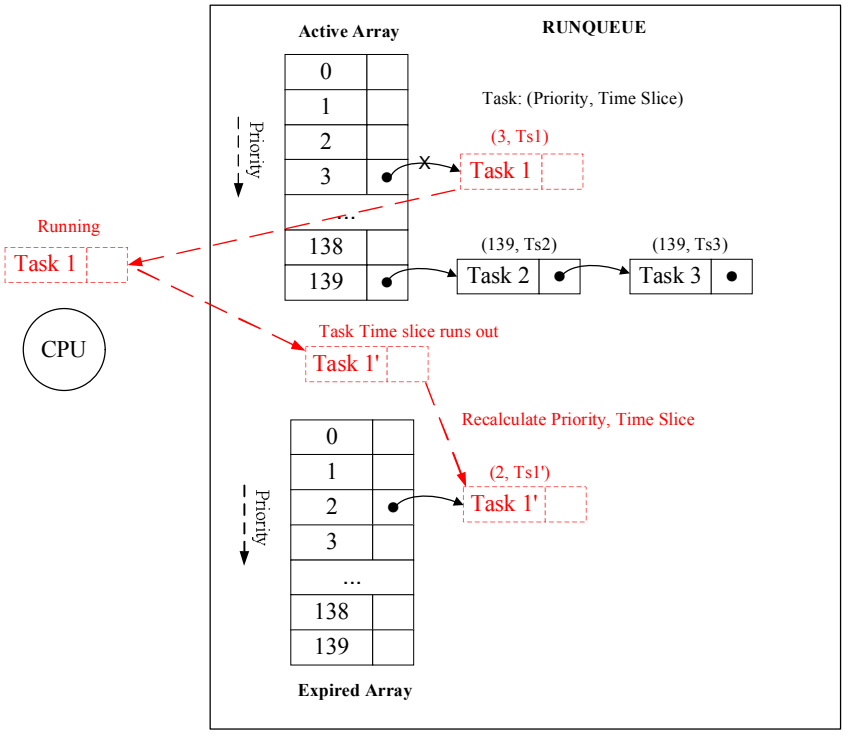

Figure 1.1: Linux kernel 2.6.16 scheduler runqueue and process scheduling [CRA07]

The active array consists of 140 lists corresponding to 140 possible priorities in the system. The first 100 priorities are reserved for real time processes. Processes are added to a particular priority list from its tail, and are consumed from the list's head. The scheduler selects the next process to run from the head of the highest non-empty priority

list. When an interrupt occurs or the scheduler clock ticks, the running process will be preempted if it does not hold any kernel locks and a higher-priority task becomes available. The priority of a process consists of a sum of two values: a nice value, which is a static priority value assigned by the user to the process ranging from 1 to 40, and a bonus value, which is a dynamic priority value in the range from -5 to +5 assigned by the processor to indicate the interactivity of the process [TOR07], [LOV05].

Each process is assigned a timeslice based on its nice value, thus higher priority processes are granted longer timeslices. A process might not use its entire timeslice at once because it gets blocked for an I/O or preempted by a higher priority process. However, the process will eventually consume its entire timeslice, and then the process is placed in the expired array with a new timeslice and a recalculated priority [TOR07], [LOV05].

The scheduling of interactive tasks differs from the way described above. Although interactive tasks receive the same timeslice assigned to their peers of the same priority, their timeslices are subdivided into smaller pieces. When a subdivision of the timeslice is consumed, the process will round robin with other processes of the same priority. Thus, the execution is rotated more frequently among interactive tasks. Furthermore, when an entire timeslice of an interactive process is consumed, the process is not placed in the expired state. Rather, the process is given a new timeslice in the active array, unless processes in the expired array are starving or is having a higher-priority process. Starvation, in this case, occurs when the amount of time proportional to the number of processes in the active array has passed since the last switch between the active and expired arrays [TOR07], [LOV05].

Interactivity of processes is calculated dynamically by the bonus priority value. In general, it is difficult for a low-priority process to qualify as an interactive process, while it is difficult for a high-priority process not to qualify as an interactive process. The bonus priority is calculated based on the sleep average, which is a value indicating how long the process has been waiting (sleeping) by increasing the value while the process is waiting and decreasing the value while the process is running [TOR07], [LOV05].

1.3 Motivation

One of the most important tasks of the kernel is managing the hardware connected to the computer. Thus, the kernel needs some mechanism to communicate with the machine's individual devices. Since the processor is usually much faster than the hardware connected to the machine, it is inefficient for the processor to issue a request and waits for a response from a very slow device. Instead, the processor should be able to do something else while the hardware is preparing the answer to the processor's request. One way of implementing this is through device polling: the kernel regularly checks the status of the hardware devices for any event. However, this solution incurs overhead regardless of the status of the hardware, which might be not ready or inactive. A more efficient solution is to provide the hardware devices a way to signal the kernel for any specific event. This signaling from the hardware device to the kernel is so-called interrupts [LOV05].

In many interrupt-driven preemptive operating systems, such as Linux, interrupts are often given high priority. Thus, interrupts usually preempts the process running in the CPU and the execution is transferred to the corresponding interrupt handler. Therefore, the implementation of the way the hardware controller communicates with the kernel through interrupts is very important, since those interrupts can create a great load on the processor, leaving no chance for other processes to execute. A good example of this is the network device controller. Under normal conditions, the network controller sends an interrupt to the processor for every received packet. This is acceptable under low network traffic environment. However, when the network traffic increases, this way of communication is inefficient and can easily lead to a livelock condition [MUR06]. Thus, the way in which interrupts are originated and scheduled is very crucial for system performance.

In [QAH07], a study is conducted by Salah and Hakim to compare different algorithms for interrupt handling in the network device controller, in which they measured the CPU availability under variable network loads. This measurement was taken using the Simplex application, which is a CPU-bound user application that can run for a specific time interval on a CPU. This time interval will increase when Simplex runs on a busy CPU. Their results in Figure 1.2 show strange behavior around 70-100 Kpps. The amount of CPU resources available for user applications, which is reversely proportional to the execution time of Simplex, greatly decreases starting from network load of 70 Kpps. This is clearly visible from the sharp peak in Simplex execution time around the rate of 80 Kpps. Since any increase in the network traffic arrival rate on the recipient end would cause a demand for more CPU resources, the trend of the graph in

Figure 1.2 should be, logically speaking, a monotonically decreasing trend, as the amount of CPU resources left for user applications should decrease as the network traffic increases. However, the results in [QAH07] show that after the 100 Kpps traffic rate, the amount of CPU resources that is available to user applications sharply increases. This can be clearly deduced from the sharp decrease in Simplex execution time beyond 100 Kpps. As network traffic rate continues to increase beyond 100 Kpps, the amount of CPU resources assigned to Simplex gradually decreases, where Simplex execution time increases slowly.

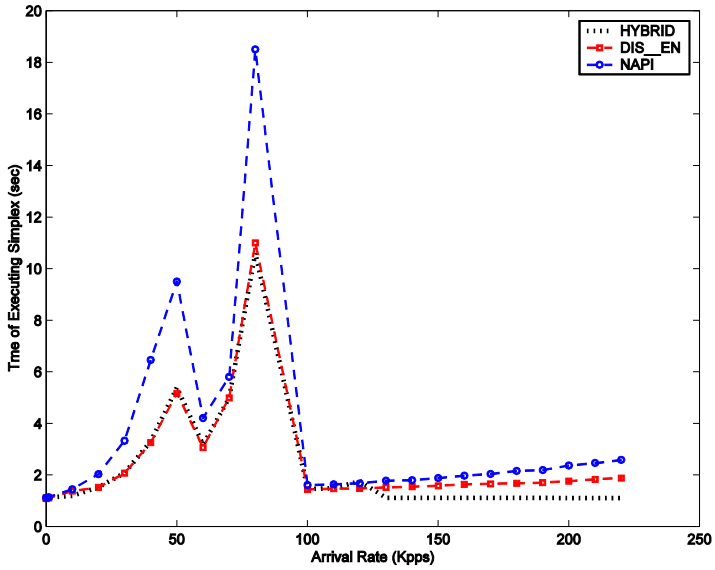

Figure 1.2: Execution time for Simplex application [QAH07]

In their explanation of this behavior, Salah and Hakim interpreted this strange behavior around 70-100 Kpps as a consequence of the reduction in the interrupt-handler dynamic priority level in this network traffic range. They stated that the drop in the dynamic priority level results from the fact that the interrupt handler at this high traffic

rate consumes its entire timeslice, where the scheduler (Linux 2.6 O(1) Scheduler, in particular) decrease its bonus value, and thus its dynamic priority. As a result, the user applications, Simplex in this case, get more chance to run, as its priority becomes equal or even higher than the priority of the interrupt handler.

In Linux, frequent interrupts caused by received or transmitted packets may stretch the execution and eventually starve user processes. This problem gets exacerbated in the presence of network I/O-bound processes, which do not often consume their entire CPU quantum. Thus, they are given higher priorities than CPU-bound processes, which consume their entire CPU quantum and get demoted. Starvation of CPU-bound processes was previously experimentally demonstrated in [QAH07], whereby a user process starves as its execution time escalated to 30 to 60 folds under normal network load. In this research work, we aim to assess and study the issue further under various system and network configurations. Then, we shall propose and implement simple solutions to the observed starvation issue under different versions of Linux. The performance of those solutions shall be analyzed and evaluated experimentally. Finally, we shall come up with a more comprehensive solution approach that completely alleviates this starvation problem.

In [CRA07], it has been shown that Linux interactivity mechanism can lead to a serious starvation issue as it tends to incorrectly categorize non-interactive network applications as interactive, such that they can unjustifiably obtain up to 95% of the CPU. In [CRA07], Wu and Crawford have studied network processes at relatively high network loads (i. e. more than 100 Mbps). In addition, they did not show how the Linux scheduler deal with network processes as the network traffic rate gradually increases.

Moreover, Wu and Crawford have only studied this starvation issue in Linux 2.6 O(1) Scheduler. Additionally, the solution to the starvation issue proposed in [CRA07] is a global solution, where a single global minimum threshold is used to filter out the sleeps of all processes in the system. It is also worth noting that Wu and Crawford have only dealt with uni-processor network recipient systems in [CRA07].

On the other hand, in this research work we study the behavior of the network processes at relatively normal network loads, not exceeding 100 Mbps. In addition, we show in detail the behavior of the Linux scheduler in dealing with network processes as the network traffic rate gradually increases, which shows that there is a certain traffic rate at which the network processes become "CPU-hogs" and starve other processes in the system. Moreover, we study the starvation issue under two different versions of the Linux scheduler, namely, Linux 2.6 O(1) scheduler and Linux 2.6 CFS, where we show analytically and experimentally that the starvation issue is also present under Linux 2.6 CFS. Furthermore, we propose a more comprehensive solution to the observed starvation issue, where a localized per-process threshold is used to limit the sleeping behavior of the network process. Finally, we also study the starvation issue and the effectiveness of the introduced solutions under symmetric multiprocessor (SMP) configuration.

1.4 Main Contribution

Linux is a well-know widely used operating system. Therefore, improving the performance of Linux is a significant contribution to the public computer community. In

addition, as computer networks become handy nowadays, the need to enhance system performance while processing network I/O becomes more urgent. In general, this research work will contribute in the following areas:

- **Experimental Assessment of CPU-Bound Processes Performance under Different System Configurations.** In this research work, a detailed experimental study of the performance of CPU-bound processes in Linux shall be carried out. Different types of CPU-bound processes, such as Simplex, are to be exploited to carefully selected system configurations and their performance is then recorded and analyzed. This assessment shall provide a significant enlightenment to the performance issue faced by CPU-bound processes while processing network I/O in Linux.

- **Analysis of CPU-Bound Processes Performance under Two Different Versions of Linux Scheduler.** After the performance issue of CPU-bound processes is experimentally analyzed and thoroughly assessed, a complete causal analysis shall be carried under both versions of the Linux process scheduler, namely, the Linux 2.6 O(1) Scheduler and the Linux Completely Fair Scheduler (CFS). This analysis shall lead to identification of the root cause of the performance issue at hand. This shall help in proposing potential solutions.

- **Proposing Potential Solutions.** After the identification of the root cause of the observed performance issue of CPU-bound user processes, we will propose a set of potential solutions. Two solutions approaches are proposed for both versions of the

Linux scheduler, namely, a simple global solution and a general local per-process solution. The simple solution serves as a proof-of-concept; therefore, it shall be implemented and tested. After that, a more involved general solution approach shall be proposed.

- **Exploring the Performance Issue under SMP.** A separate analysis of the problem under symmetric multiprocessing environments (SMP) shall be carried out. The extent of the problem under SMP shall be determined under both versions of the Linux process scheduler. The effectiveness of the proposed solutions under SMP shall be also studied.

1.5 Organization of Thesis

This thesis is organized as follows. Chapter 2 gives a literature survey of both the history and the previous work related to Linux process scheduler. Chapter 3 presents an experimental assessment of the performance of CPU-bound processes under different system configurations and versions. Chapter 4 presents a detailed analytical study of the performance issue. Chapter 5 presents proposed solutions. Chapter 6 presents a study of the performance issue under symmetric multiprocessing (SMP) environment. Chapter 6 gives the conclusion and future work.

CHAPTER 2

LITERATURE REVIEW

2.1 Introduction

As the Linux is an open-source operating system, it is being continuously modified, improved and repaired. In this chapter, we give an overview over the literature related to the Linux process scheduler. We start first by giving a brief overview over the evolution of the Linux process scheduler and the stages it has passed through. Then, we present a survey of the work related to the Linux process scheduler.

2.2 Evolution of the Linux Process Scheduler

Throughout the life of Linux, its process scheduler has undergone a series of major changes that contributes to its scalability, fairness, and interactivity handling. In fact, the history of the Linux process scheduler can be divided into three main stages, ordered by time as follows: the Linux 2.4 O(n) Scheduler, the Linux 2.6 O(1) Scheduler, and the Linux 2.6 CFS. We will give a brief idea about each stage.

The Linux 2.4 O(n) Scheduler was the original Linux scheduler. Its design was very simple, which uses the goodness() function to recalculate the priority of each task at every context switch to determine the next task to switch to. Its name, i. e. O(n), stems

from its algorithm complexity, which is a linear complexity. This scheduler design has not changed a lot through the successive releases of the 2.4 version of Linux. However, major limitations of this scheduler started to get more and more apparent. Good examples include its linear complexity, its inability to scale well, and its lack of SMP utilization when the number of systems processors increases. Thus, by 2001, the Linux user community started to get more serious calls for change, see [KRA01] for an example [LKA].

Later, in January 2002, the Linux O(1) Scheduler was firstly introduced to the 2.5 kernel series by Ingo Molnar. The scheduler has a new unique design, in which each processor has a separate runqueue consisting of two different arrays: an active array and an expired array. The active array holds ready tasks, while expired tasks are put in the expired array. As the name describes, the O(1) Scheduler chooses the task to run next on the CPU in constant time, no matter how many tasks are there in the runqueue. Thus, it is mostly noted for its excellent scalability [LKA].

In the period between 2002 and 2006, the O(1) Scheduler was extensively developed. In 2002, there has been a tremendous amount of work on the scheduler regarding many scheduling issues, such as preemption, user mode Linux support, runtime tuning, non-uniform memory access (NUMA) support, CPU affinity, scheduler hints, 64-bit support, and gang scheduling. By the end of 2002, the O(1) Scheduler has become the standard even in the 2.4 kernel series. Between 2003 and 2004, the concentration was more on the scheduler interactivity problems and hyperthreading. Proposals for a new scheduler started to appear in 2004, such as Nick Piggin's domain-based scheduler and Con Kolivas' staircase scheduler [LKA].

In 2007, Con Kolivas proposed a new scheduler: The Rotating Staircase Deadline Scheduler (RSDS). This scheduler proposal proved that fair scheduling can be actually implemented. This has inspired Ingo Molnar, the original O(1) Scheduler, to come up with a new scheduler, which he named as the Completely Fair Scheduler (CFS). This scheduler has been merged into kernel 2.6.23 to become the new Linux scheduler [LKA].

2.3 Related Work

Recently, a considerable amount of research has been dedicated to Linux process scheduler improvement. The scheduler is studied from different prospective, and several approaches are used to enhance its performance. A special emphasis in the literature has been towards improving the scheduler's interactivity and fairness. In this section, we present some of the recent important studies related to the Linux process scheduler.

In [WU99], Wu and Kuo explored real-time scheduling of processes which may share non-preemptible resources on the same processor and, at the same time, request services from other independent subsystems. They considered the scheduling of real-time processes which may stop to wait for disk I/O without releasing any locked semaphores. They proposed a revision of the definition of priority ceiling and a resource synchronization mechanism to meet the deadline requirements of real-time CPU-bound and I/O-bound processes.

In [ETS03], two solutions to the problem of having less support from current systems for interactive applications, especially multimedia applications, were proposed.

The first was to devise specialized scheduling mechanisms that take the specific needs of such applications into account. The second solution was to tune the existing system by increasing the clock interrupt rate, which will make the scheduler information about CPU usage more accurate.

In the last few years, Linux scheduler has been greatly modified. In fact, one of the major changes in the kernel of Linux 2.6, released to the public in December 2003, was an entirely new scheduling algorithm. This scheduling algorithm was designed to achieve the following new features:

- A bound of O(1) on the time needed to choose the next process to run.
- Quick response to interactive processes under high system load.
- An acceptable level of prevention of both starving and hogging.
- Scalability and task affinity under symmetric multiprocessing environment.
- Improved performance when having a small number of processes [TOR07], [LOV05].

Moreover, the kernel in Linux 2.6 became preemptible, as opposed to the prior releases of Linux, which have non-preemptible kernels [WU07].

The Variable-Rate Execution (VRE) model was applied to Linux scheduler to support dynamic Quality of Service (QoS) in [LIU04]. Their VRE scheduler can assign a specified execution rate to any application, and dynamically adjust the execution rate during runtime. They used rate controller components to adjust task's execution rate based on predefined rules and runtime feedbacks, such as suspension time, the queue length, etc.

In [ETS04], the common prioritization of processes for scheduling based on their CPU usage was identified to be inaccurate in many cases, such as interactive and multimedia applications. A proposed solution was to directly quantify the I/O between an application and the peripheral device, such as the keyboard, the mouse or the screen.

In [ZOL04], a dynamic scheduling algorithm with minimum context switches was used for spacecraft avionics systems. The algorithm is a modified version of the well-know Minimum Laxity First (MLF) algorithm, which is optimized for spacecraft avionics systems. Their mathematical analysis and simulation shows that this proposed algorithm imposes less context switches and is more applicable to spacecraft avionics systems.

A visualization tool for the Linux processor scheduling algorithm was introduced in [LEU05]. The tool can be used to simulate several scheduling scenarios and it provides different reports. In addition, the tool can be extended to serve as the basis for process performance measuring or optimization purpose.

In [TOR07], a simple multilevel feedback queue (MLFQ) scheduler was implemented in the Linux 2.6 and compared to the new Linux 2.6 scheduler with respect to response time to interactive tasks. Two different algorithms based on MLFQ were implemented and compared to the Linux 2.6 scheduler. Their results show that MLFQ scheduler is comparable to Linux 2.6 scheduler in terms of response time, and yet has better turnaround time than Linux 2.6 scheduler.

The effects of having a preemtible kernel in Linux 2.6 on the transmission control protocol (TCP) networking were studied in [WC07]. The results from both mathematical modeling and practical experiments show that having preemptible kernel

can interact badly with the performance of the networking subsystem and imposes some performance bottlenecks in Linux TCP.

In [TSA07], the overhead of context switch inflicted by periodic clock interrupts were measured by two methodologies: changing the interrupt frequency while measuring the time taken to sort an array, and measuring the time taken to execute a simplistic loop calibrated to run for 1ms. The results of two methodologies are different.

In [SAL07], a mathematical analysis and discrete- event simulation are used to study and compare the performance of various proposed interrupt handling schemes in gigabit networks, as the performance of gigabit network end hosts or servers can be severely degraded due to interrupt overhead caused by heavy incoming traffic. A novel hybrid scheme of interrupt disabling–enabling and pure polling is also proposed.

In [CRA07], it is shown that Linux interactivity mechanism of Linux 2.6 O(1) Scheduler tends to incorrectly categorize non-interactive network applications as interactive, which can lead to serious fairness or starvation issues. A solution to this interactivity vs. fairness problems is also proposed.

In [WON08], it is shown that Linux Completely Fair Scheduler (CFS) has a fairness issue in multi-threaded environments. The study shows that CFS tends to favor programs with higher number of threads, as the CFS algorithm is based on thread-level fair scheduling. A novel algorithm is proposed to achieve better fairness in the Linux CFS.

CHAPTER 3

PERFORMANCE ASSESSMENT OF CPU-BOUND PROCESSES UNDER DIFFERENT NETWORK CONFIGURATIONS

3.1 Introduction

In Linux, frequent interrupts caused by received or transmitted packets may stretch the execution and eventually starve user processes. This problem gets exacerbated in the presence of network I/O-bound processes, which do not often consume their entire CPU quantum. Thus, they are given higher priorities than CPU-bound processes, which consume their entire CPU quantum and get demoted. Starvation of CPU-bound processes was previously experimentally demonstrated in [QAH07], whereby a user process starves as its execution time escalated to 30 to 60 folds under normal network load.

In order to thoroughly assess the problem and quantify its scope, the performance of the system has to be studied under different configurations as well as different types of network I/O-bound processes. In this chapter, the performance problem is assessed with different types of network I/O-bound processes, different network adapters (NIC's), and different versions of the Linux Kernel. The results show that the problem does exist

under all these configurations. The remainder of this chapter is organized as follows: Section 3.2 illustrates the experimental setup used for conducting all the experiments, Section 3.3 gives an overview over the performance metric used to measure the performance of the network process. Then, the study of system performance with different network I/O-bound processes is presented in Section 3.4. After that, Section 3.5 discusses the system performance with different types of NIC's. Finally, Section 3.6 analyzes the system performance under different versions of the Linux Kernel.

3.2 Experimental Setup

In all of the experiments carried out for assessing the problem scope, we use two Linux machines of a sender and a recipient connected to each other via 1 Gbps Ethernet crossover cable (Figure 3.1). Both machines use Fedora Core 5 Linux with kernel 2.6.16. The sender machine runs on Intel Xeon CPU (3.6 GHz) with 4GB RAM and a network adapter with BCM 5751 network controller. The recipient machine runs on Intel Pentium 4 CPU (3.2 GHz) with 512MB RAM and a network adapter with BCM 5752 network controller. To minimize the impact of other system activities on performance and measurement, we boot up both machines with run level 3, and we make sure that no services are running in the background. We also disabled Ethernet link flow control.

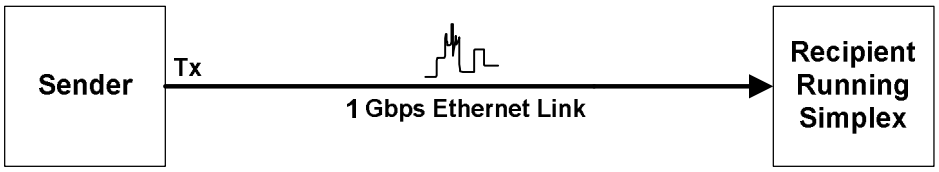

Figure 3.1: Experimental Setup

In order to measure the CPU availability for user applications under variable network loads, Simplex application was utilized. Simplex is a CPU-bound user application that can run for a specific time interval on a CPU. This time interval will increase when Simplex runs on a busy CPU. The network traffic was varied from 0 to 150000 packets per second (150 Kpps).

3.3 Performance Metrics

There are several quantities associated with each process that can be used as a gauge to the process performance. These quantities include the process's execution time, system (kernel) time, user time, involuntary context switches, and CPU utilization. We use all of those quantities as a representative measure of the process (i. e. Simplex in this research work) performance in the presence of a number of network processes and under various system and network configurations. To measure those parameters, we use the *time* utility of Linux. The time utility runs a command with its arguments, if any, and then returns statistics about the running time of the command, which consists of the elapsed time, the user time and the system time. The GNU version of the *time* utility, when run in the verbose mode or using the output format string option, gives much more details about the program performance including different time, memory, I/O and IPC measurements. Thus, we used the GNU version of the *time* utility with the verbose mode to measure the following performance metrics of Simplex application: the execution time, the user time, the kernel time, the number of involuntary context switches, and the

percentage of the CPU resources the process was able to get. The syntax used to run the time utility in the verbose mode is (in Korn Shell):

```
$ time –v ./Simplex Simplex_time
```

Where `Simplex_time` is the amount of time Simplex application calculations should take on a free CPU. The performance metrics used to measure Simplex performance throughout this research work are described in detail below.

- **Simplex Execution Time.** (also called *elapsed time*) The time Simplex takes to complete all of the calculations and exits. This time approximately equal to the parameter fed to Simplex if the CPU is totally free, or if Simplex get 100% of the CPU resources. However, this amount of time increases when Simplex get less CPU resources due to high load on the system or unfair scheduling of Simplex. This quantity is represented by the *%E* character in the output format string of the *time* utility.
- **System (Kernel) Time.** (also called *stime*) The amount of CPU time the process (i. e. Simplex) spends in kernel mode. It is represented by *%S* in the output format string of the *time* utility.
- **User Time.** (also called utime) The amount of CPU time the process (i. e. Simplex) spends in user mode. It is represented by *%U* in the output format string of the *time* utility.
- **CPU Availability (or Percentage).** The percentage of CPU resources the process (i. e. Simplex) got. It is computed as the ratio of the sum of *system time* and *user tim*e to the total *elapsed time*, i. e.:

(system time + usertime) / (Simplex execution time.)

CPU Availability is represented by *%P* in the output format string of the *time* utility.

- **Number of Involuntary Context Switches.** The number of times the process (i. e. Simplex) was preempted, or forced to give away the CPU resources involuntarily, either due to timeslice expiration or the presence of a ready task with a higher priority value than Simplex at any scheduler tick (i.e. interrupt). The number of involuntary context switches is represented by *%c* in the output format string of the *time* utility.

3.4 Performance under Different Network I/O-bound processes

Among the most important factors that need to be varied when studying the scope of the observed performance problem are the type and number of network I/O-bound processes running at the recipient side. There is a wide variety of available network traffic analysis tools, which corresponds to different types of network I/O-bound processes. Some network traffic generation tools, such as the open source *Distributed Internet Traffic Generator* (D-ITG), comes with their own traffic analysis tool, namely *ITGRecv*. On the other hand, other network traffic generation tools does not offer such a utility, and thus the traffic has to be captured and analyzed by other network traffic analysis tools.

To account for both types of network traffic generation tools, three types of network I/O-bound processes were studied. First, *ITGRecv* utility, which comes with D-ITG, was used as a network I/O-bound process that captures a traffic generated by D-ITG. On the other hand, *tcpdump* utility and *ethereal* utility were used as network I/O-bound processes that capture a traffic generated by KUTE, a kernel-based traffic engine. The following three sub-headings presents the analysis of the recipient system performance with the three types of network I/O-bound processes, namely, *ITGRecv*, *tcpdump* and *ethereal*.

- ***ITGRecv* as a Network I/O Process.** Distributed Internet Traffic Generator (D-ITG) is an open-source non-commercial tool capable of generating network traffic from the user level. In D-ITG, two daemons are used: one is for sending the traffic at the sender side, and the other one is for receiving the traffic at the recipient side. We have installed D-ITG 2.4.4 to conduct the experiment. The traffic was varied from 0 to 150 Kpps. The packet carries 64 bytes in its payload. On the recipient side, Simplex was triggered to run for 10 seconds at each 10-Kpps-step of the traffic rate. The actual time Simplex took to complete is shown in Figure 3.2.

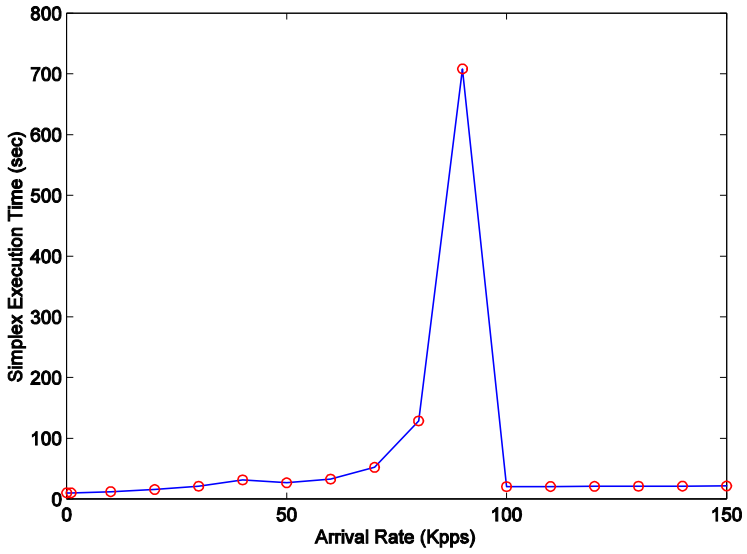

Figure 3.2: Performance measurement with *ITGRecv* as a network I/O-bound process

As is apparent from the figure, there is a sharp jump of Simplex time around the rate of 90 Kpps. In fact, at 90 Kpps, the Simplex user process took more than 700 seconds to complete calculations that should not take more than 10 seconds given free CPU resources. This clearly indicates that the performance problem at hand does exist with *ITGRecv* as a network I/O-bound process.

- ***Tcpdump* as a Network I/O-bound process.** KUTE version 2.4 was used to send the traffic from the sender side, while *tcpdump* utility was used to capture the generated traffic. KUTE, the kernel based traffic engine, is a freely distributed traffic generation tool that is built in the form of a kernel module that is able to send UDP packets at the kernel level. The main advantages of kernel-based traffic generation tools, such as KUTE, is that they do not suffer from the frequent context switching the user-level traffic generation tools usually suffer from, and their increased accuracy in measuring the inter-packet timing.

At the recipient side, *tcpdump* utility was used to simulate the network receiving activity. Tcpdump was run using the command line option and was logging the captured traffic to files in order to avoid any GUI or display overhead, as follows:

```
$tcpdump -C 1000 -W 2 -w /var/tmp/tcpdump.out
```

Simplex was triggered to run for 10 seconds at the recipient side, and was analyzed using the *time* utility.

The results of this experiment are shown in the Figures 3.3(a) to 3.3(e). In line with the previous results of D-ITG, the total time Simplex took to complete shows a distinguished peak at the rate of 90 Kpps (Figure 3.3(a)). With a closer look at Figure 3.3(a), we find a small jump around the rate of 50 Kpps. This jump is due to the overhead of interrupts, which increases until the rate of 60 Kpps, where their number drops and thus Simplex get more portion of the CPU resources [QAH07]. Beyond the rate of 60 Kpps, the execution time of Simplex increases, until it reaches its maximum at the rate of 90 Kpps. This jump is clearly viewable, as Simplex took more than 600 seconds to execute. After that, Simplex time drops down, and then increases gradually as the network traffic increases.

In addition, both portions of the Simplex total time: the time it took at the kernel level, and the time it took at the user level, have a clear peak at the rate of 90 Kpps (Figure 3.3(b) and 3.3(c)). In Figure 3.3(b), the time Simplex took at the user level, i. e. user time, have a small peak around the rate of 50 Kpps and a larger peak at the rate of 90 Kpps. However, since the time Simplex spends at the user level is relatively small compared with the time it spends at the kernel level, the difference between the two peaks is not very large. The time Simplex spends at the kernel level, i. e. system time,

also show two peaks, one at the rate of 50 Kpps, and a larger one at the rate of 90 Kpps. The difference between the peaks is large, showing a clear performance bottleneck at the rate of 90 Kpps.

Figure 3.3(d) shows the number of involuntary context switches Simplex application was forced to make versus different traffic rates. An involuntary context switch occurs when either a higher priority process has taken over the CPU or when the process has used up an entire timeslice and had its priority reduced as a consequence. Involuntary context switches indicate that there is some contention for the CPU, and a bottleneck exists. The figure shows that the number of involuntary context switches was maximum at the rate of 90 Kpps, reaching almost 6 million context switches (Figure 3.3(d)). In addition, there is a small peak of involuntary context switches around the rate of 50 Kpps, which is due to the increase in the number of interrupts around this rate, as they have higher priority than the user CPU-bound process, Simplex.

Finally, the percentage of CPU resources available for the Simplex application is shown in Figure 3.3(e). As appears from the graph, the percentage of CPU resources Simplex was granted is minimal at the rate of 90 Kpps, magnifying the performance issue. Moreover, there is a local minimum at the rate of 50 Kpps, where the interrupts are maximum, and thus take most of the CPU resources.

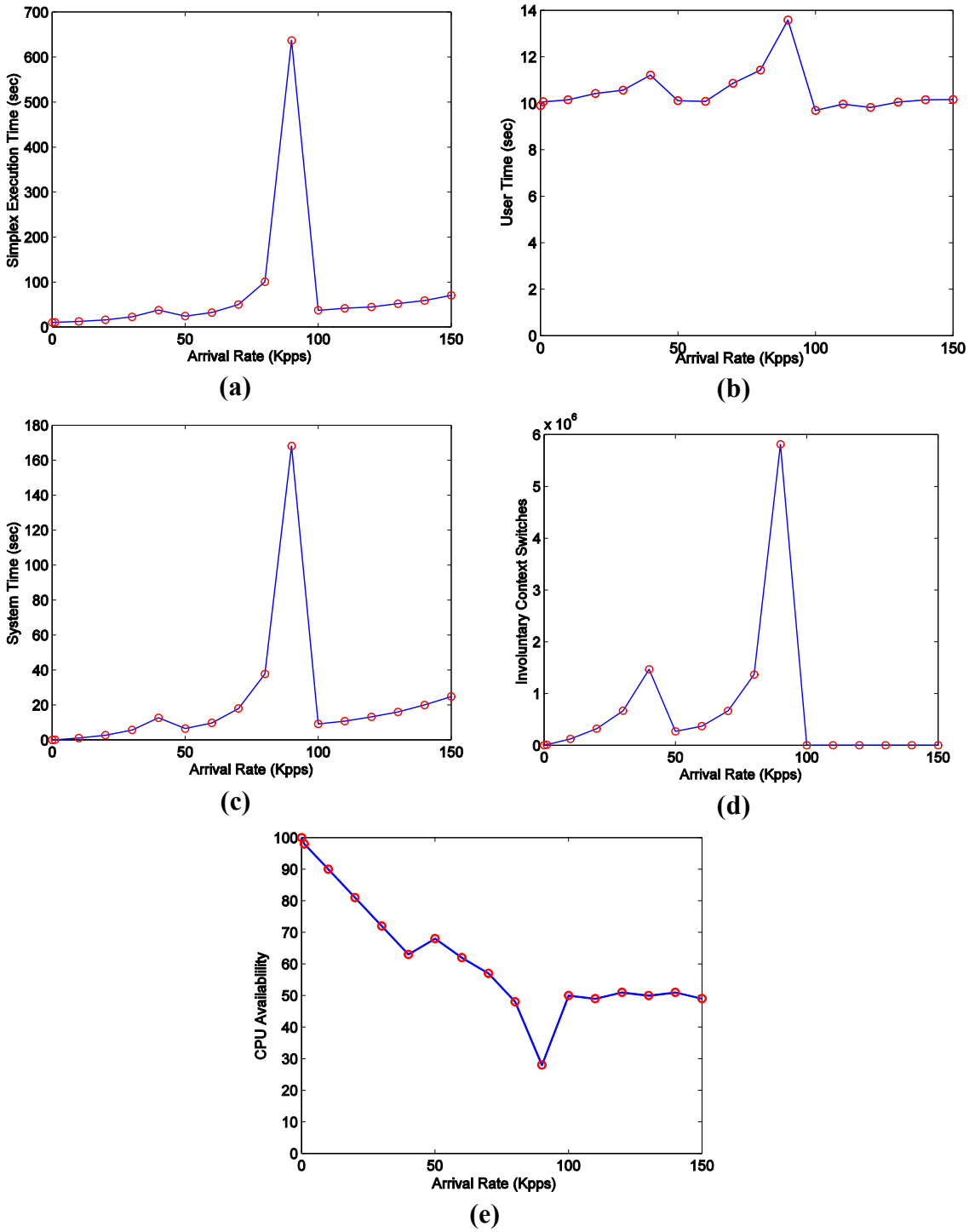

Figure 3.3: Performance measurement with *tcpdump* as a network I/O-bound process

- ***Ethereal* as a Network I/O-bound process.** The results of repeating the previous experiment with a different network traffic capturing tool, namely, Ethereal, are shown in Figures 3.4(a) through 3.4(e). The results are in perfect harmony with the previous experiment results, showing a sharp increase of Simplex time, user time, system (kernel) time, and number of involuntary context switching around the rate of 90 Kpps. Moreover, the CPU availability for Simplex is minimal at the rate of 90 Kpps, as was the case of the previous experiment involving tcpdump.

In Figure 3.4(a), the Simplex time is plotted against the network traffic arrival rate. The Simplex time increased to a local maximum before the rate of 50 Kpps, then decreases. This local maximum in Simplex time is caused by the increased number of interrupts at this rate. After that, Simplex time decreases to a local minimum at the rate of 50 Kpps, then increases sharply to an absolute maximum at the rate of 90 Kpps. In fact, Simplex takes around 300 seconds to execute calculations that would not take more than 10 seconds on a free CPU. This absolute maximum indicates that there is an apparent performance bottleneck at the rate of 90 Kpps. After the rate of 90 Kpps, Simplex time drops down, and then increases gradually with the increasing rate of network traffic.

Figure 3.4(b) shows the time Simplex spends at the user space, plotted against network traffic arrival rate. The graph shows two maximum points around the rates of 40 Kpps and 90 Kpps. The first maximum point, as was the case of tcpdump, is due to the increasing number of interrupts at this rate [QAH07]. After that, the interrupt rate drops down and Simplex gets more chance to execute, thus its time drops down. However, at

the rate of 90 Kpps, the user time of Simplex increases to another maximum point. Since the time Simplex spends at the user space is not very large, the difference between the two maximum points is very small. However, it is clear that there is a performance issue at the rate of 90 Kpps. After 90 Kpps, the time Simplex spends at the user space is almost constant, which is approximately 10 seconds.

The time Simplex spent at the kernel level is plotted against the arrival rate in Figure 3.4(c). The graph is very similar to the graph of Simplex total time versus arrival rate in Figure 3.4(a). This is reasonable since the kernel time comprises most of the total time Simplex takes to execute. Figure 3.4(c) shows a small jump in Simplex system time at the rate of 40 Kpps, then a very sharp jump in Simplex system time at the rate of 90 Kpps, which clearly highlights the performance bottleneck at this rate. Beyond 90 Kpps, Simplex time drops down and then increases proportionally with the increase in network traffic arrival rate.

In accordance with other results, Figure 3.4(d) shows that the number of involuntary context switches Simplex was forced to make are maximum at the rates of 40 Kpps (locally) and 90 Kpps (globally). The first maximum point is due to the increased number of interrupts at this rate, while the second absolute maximum point is a clear evidence of a performance bottleneck at the rate of 90 Kpps. At 90 Kpps, the number of involuntary context switches reaches up to 2.5 million, magnifying the performance issue at this rate.

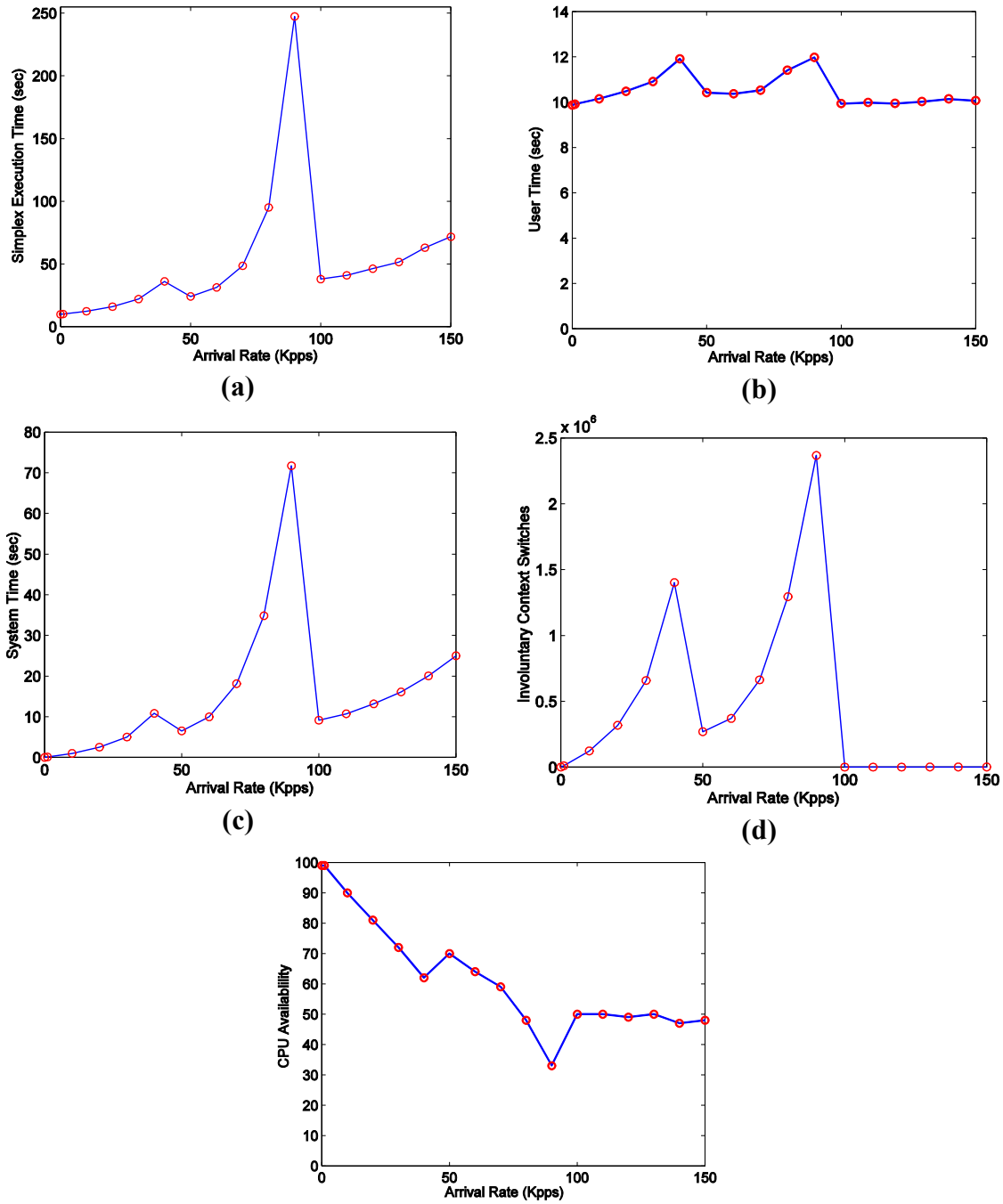

Figure 3.4: Performance measurement with *ethereal* as a network I/O-bound process

Finally, Figure 3.4(e) depicts the percentage of CPU resources available for Simplex at different network traffic rates. At the rate of 40 Kpps, there is a local minima in the graph, where Simplex could not get much of the CPU resources due to the increased number of interrupts, which have more scheduling priority than Simplex. However, at the rate of 90 Kpps, the percentage of CPU Simplex was granted is at an absolute minimum, reaching down to 30%. This shows a clear performance problem at this rate, were Simplex process was almost starving. After 90 Kpps, the percentage of CPU resources available for Simplex is almost constant, ranging around 50%.

Besides varying the type of the network I/O-bound process, the number of network I/O-bound processes might have an effect on the observed starvation phenomenon. To study the effect of the number of network I/O-bound processes on the starvation phenomenon, we investigated the performance of Simplex on a recipient machine running two *ITGRecv* processes. Figure 3.5 shows the execution time of Simplex that runs on a recipient system having one and two *ITGRecv* running on the background. The results in Figure 3.5 show that the performance of Simplex gets much worse when running multiple *ITGRecv* processes, as Simplex execution time around 80 Kpps jumps from almost 70 seconds when running one *ITGRecv* process to 450 seconds when running two *ITGRecv* processes. Thus, as the number of network I/O-bound processes on the recipient side increases, the starvation phenomenon is more exaggerated.

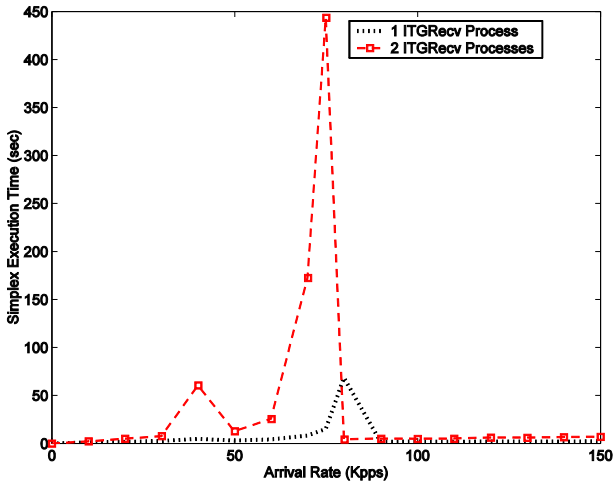

Figure 3.5: Performance measurement with multiple network I/O-bound processes

The This similar behavior of Simplex with different types and numbers of network I/O-bound processes indicates that the performance problem at hand is not dependent on the type or number of the network I/O-bound process. The problem is present under different types of network I/O-bound processes, including *ITGRecv*, *tcpdump* and *ethereal*, and different numbers of network I/O-bound processes. Thus, regardless of the type or number of network I/O-bound process running at the recipient side, CPU-bound user processes face a clear performance degradation around the rate of 90 Kpps.

3.5 Performance under Different Network Interface Cards

In order to verify that there is no relation between the observed performance problem and the type of network interface card used, the performance was assessed under different NIC's. The NIC that was used for all the other experiments was

Broadcom BCM NIC (NetXtreme BCM5752 Gigabit Ethernet Card). In this experiment, Intel NIC (82557/8/9 [Ethernet Pro 100]) was used. The results of running Simplex application for 1 second while generating network traffic with D-ITG varying from 0 to 150 Kpps is shown in Figures 3.6(a) through 3.5(e). Similar to the experiments carried out with BCM Broadcom NIC, the results of this experiment shows a peak of Simplex time at the rate of 90 Kpps, user time and system (kernel) time around the rate of 90 Kpps (Figures 3.6(a) to 3.5(c)). However, there is no clear peak of involuntary context switches at the rate of 90 Kpps (Figure 3.6(d)). Moreover, the CPU availability does show a clear minimum at the rate of 90 Kpps (Figure 3.6(e)). This remainder of this section provides a detailed analysis of the results of this experiment.

Figure 3.6(a) shows the relation between the time Simplex takes to execute in seconds and the network traffic rate in Kpps. The graph shows a small peak at the rate of 40 Kpps, which is caused by the overhead of the increasing number of interrupts at this rate. As the network traffic rate increases, Simplex time drops down, and then increases sharply at the rate of 90 Kpps, where a clear absolute maximum of Simplex time is observed. Simplex is supposed to take only 1 second to run on a free CPU; however, it takes more than 70 seconds to execute at the rate of 90 Kpps. This clearly highlights the performance bottleneck the CPU-bound user processes suffers from at this rate. Beyond the rate of 90 Kpps, Simplex time is almost constant, ranging around 50 seconds.

In Figure 3.6(b), the user portion of the Simplex time is graphed against the incoming traffic rate. Though the user time is very small, between 0.5 and 3 seconds, its behavior does show a reasonably clear trend. Between the rates of 0 Kpps and 40 Kpps, the user time is fluctuating between 0.5 and 1.5 seconds. At the rate of 40 Kpps, the user

time is at a local peak of around 1.6 second, which is caused by the overhead of interrupts. After that the user time drops down to 1 second, and then increases to a global maximum at the rate of 90 Kpps, reaching around 2.7 Kpps. After that, the user time drops down to 2 seconds, and continues to vary slightly around this value.

Figure 3.6(c) shows the relation between the kernel portion of the Simplex time and the incoming traffic rate. As the kernel portion comprises a large part of the Simplex time, its trend is pretty much similar to the trend of the overall Simplex time, having two jumps, local one at 40 Kpps, and absolute one at 90 Kpps. The local jump at 40 Kpps is due to overhead of interrupts. However, the absolute jump around 90 Kpps is not due to overhead of interrupts, as the number of interrupts decreases beyond 40 Kpps. Thus, this jump shows that Simplex is suffering from limited CPU resources at this rate. In fact, Simplex took more than 30 seconds at the kernel side. Beyond 90 Kpps, the kernel portion of Simplex is almost constant, with a value of close to 20 seconds.

The number of involuntary context switches Simplex was forced to make is plotted against the incoming arrival rate in Figure 3.6(d). The plot shows a sharp increase in the number of involuntary context switches at the rate of 40 Kpps, reaching a value up to 800,000 context switches. The number of involuntary context switches drops down beyond the rate of 40 Kpps. There is a small jump in the number of involuntary context switches at the rate of 90 Kpps, where the number of involuntary context switches exceeds 3000. However, since this value is very small compared with 800,000, the jump is hardly viewable from the graph.

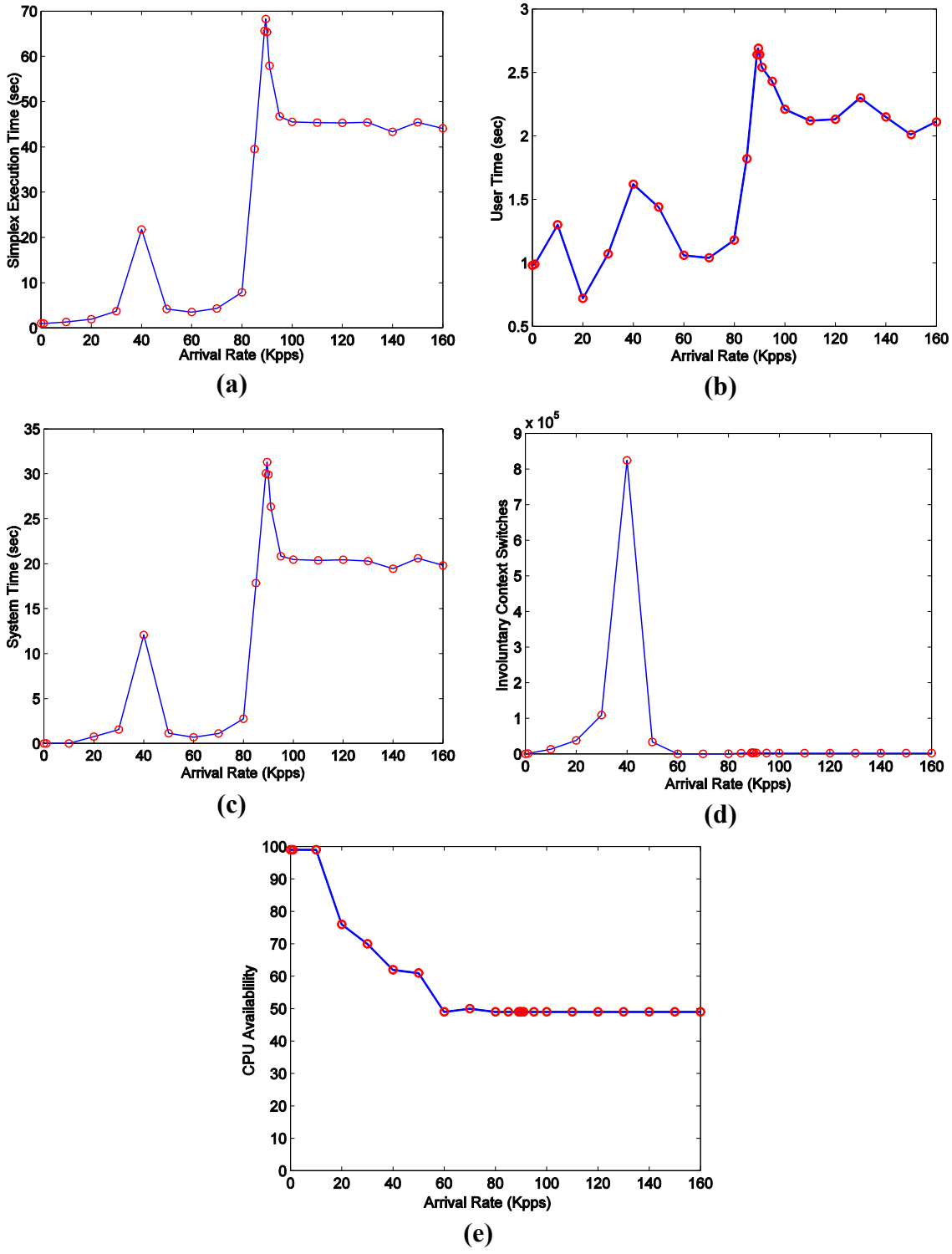

Figure 3.6: Performance measurement with Intel NIC

In Figure 3.6(e), it can be seen that the availability of CPU resources to Simplex process does not show a clear minimum neither at 40 Kpps nor at 90 Kpps. The general trend of CPU resources available to Simplex in the graph is basically a proportional decrease with the increase in the value of incoming traffic rate. Beyond the rate of 90 Kpps, the value of CPU resources available to Simplex is almost constant at the value of 50%.

The overall results of this experiment do show a similar behavior of the system under different type of NIC. However, there are some differences in the behavior of involuntary context switches as well as the availability of CPU resources. Those differences are basically due to the differences in the behavior if different NIC drivers. Nevertheless, it can be concluded that regardless of the type of NIC used, user applications in a networked Linux environment face performance degradation around the rate of 90 Kpps.

3.6 Performance under Different Versions of the Linux Kernel.

To make sure that the performance issue under investigation is not specific to Linux Kernel 2.6.16, which all other experiments were run under, the problem was assessed under the latest version of Linux Kernel, namely, Kernel 2.6.24. The results of running Simplex application for 1 second while generating network traffic with D-ITG varying from 0 to 150 Kpps is shown in Figures 3.7(a) through 3.6(e). Similar to the experiments carried out with Linux Kernel 2.6.16, the results of this experiment shows a

peak of Simplex time, user time, system (kernel) time, and involuntary context switches at the vicinity of the rate of 70 Kpps (Figures 3.7(a) to 3.6(d)). However, there is no clear minimum of CPU Resources available to Simplex Application at the rate of 90 Kpps (Figure 3.7(e)). The results are discussed in detail in the rest of this section.

Figure 3.7(a) shows the relation between the time Simplex takes to execute on the recipient side and the incoming traffic rate. As appears from the figure, there are two major jumps around the rate of 40 Kpps and 70 Kpps. At 40 Kpps, the jump is a local maximum that is caused by the increasing number of interrupts that leave less chance to Simplex to execute, and thus resulting in an increase in Simplex time. around 90 Kpps, there is an absolute maximum in Simplex time, where the time reaches more than 7 seconds. After 70 Kpps, Simplex time drops down and then increases linearly with the increasing traffic rate. This result is similar to the previous experiment result where Simplex was run on kernel 2.6.15, though the amount of time Simplex takes to execute at 90 Kpps is much lower than the result got in previous experiments.

The time Simplex takes at the user space is plotted against the incoming traffic rate in Figure 3.7(b). Since the user time is very small, ranging between 0.7 and 1.3 seconds, the graph is highly fluctuating. Nevertheless, there is a clear jump in the graph at the rate of 40 Kpps. There are other local jumps in the graph at the rates of 80 Kpps and 100 Kpps. Beyond the rate of 100 Kpps, the user time settle down around the value of 1 second. Due to the high fluctuations in the graph, there is no clear trend that can be interpreted from the graph.

On the other hand, Figure 3.7(c) illustrates the relation between the system time of Simplex and the incoming traffic rate. The graph shows a clear trend, where the time

increases with the increasing traffic rate, until it reaches a local maximum at the rate of 40 Kpps. The time drops down beyond 40 Kpps, and then sharply increases around until it reaches an absolute maximum at the rate of 70 Kpps. Beyond that point, the time drops down, and then increases linearly with the increasing value of the incoming traffic rate.

In Figure 3.7(d), the number of involuntary context switches that Simplex was forced to make is plotted against the incoming traffic rate. The number of involuntary context switches increases with the increasing value of the incoming traffic rate until it reaches an absolute maximum at the rate of 40 Kpps. Then, the number of involuntary context switches drops down, and then increases again, until it reaches another local maximum at the rate of 70 Kpps. Beyond 70 Kpps, the number of involuntary context switches increases linearly with the increasing number of incoming traffic rate.

The percentage of CPU resources available to Simplex application is plotted against the incoming traffic rate in Figure 3.7(e). The percentage decreases with the increasing value of incoming traffic rate, until it reaches a local minima at the rate of 40 Kpps. After this point, the percentage increases until 50 Kpps, and then drops down to a minimum point of 50% at the rate of 70 Kpps. Beyond 70 Kpps, the percentage of CPU available to Simplex stays constant at 50%.

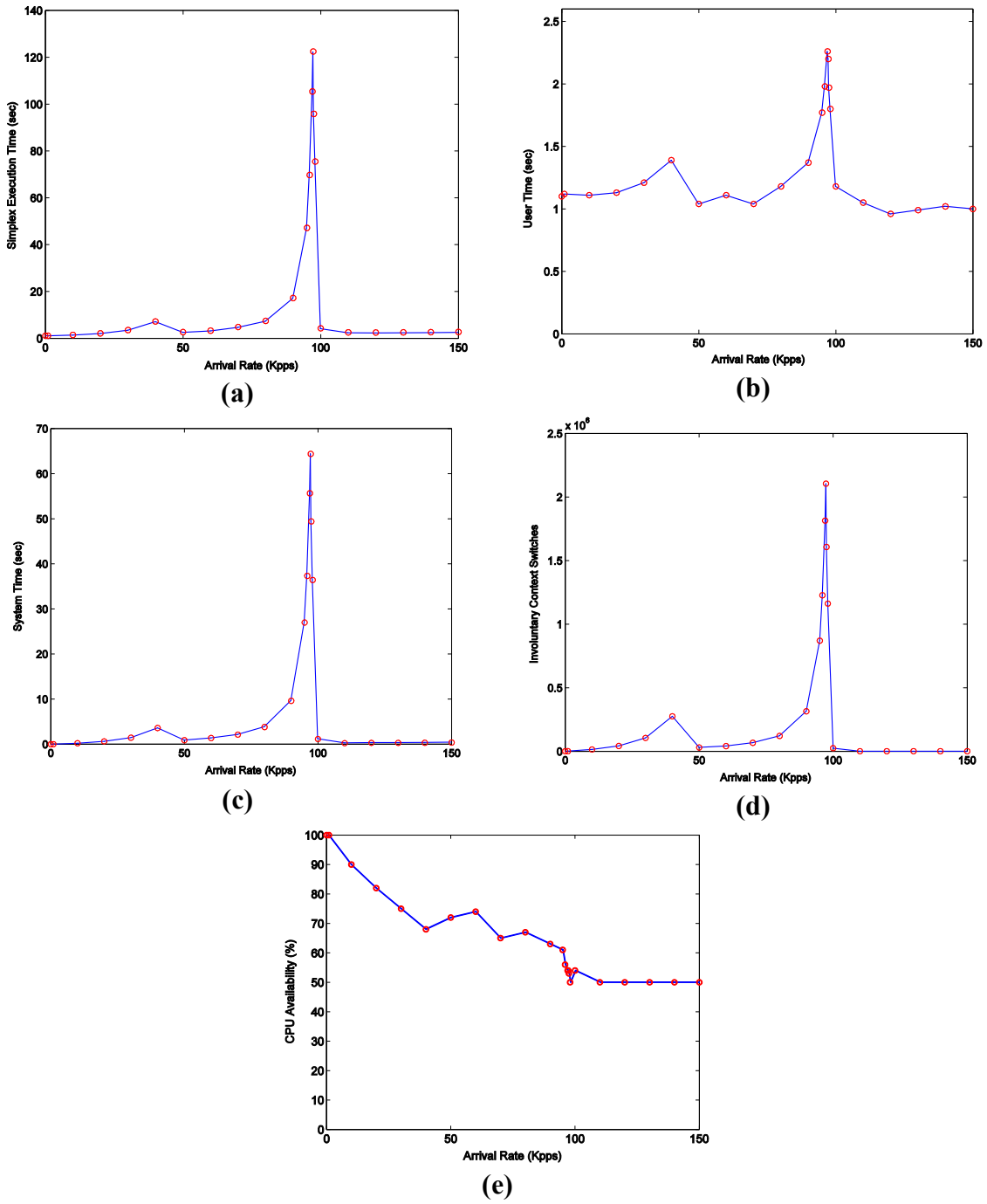

Figure 3.7: Performance measurement under Linux kernel version 2.6.24

Despite the slight differences found in this experiment, the results of this experiment confirm that the performance problem under investigation does exist in the latest kernel of Linux OS, namely Kernel version 2.6.24. In a full agreement with the results of the pervious experiments, this experiment shows that there is a clear performance bottleneck beyond the rate of 40 Kpps. In particular, the performance bottleneck appears at the rate of 70 Kpps. Though the point at which the performance degradation was 90 Kpps in the previous experiments, this difference is not significant as it is only concerns the location where the performance degradation appears, and not the existence of the problem itself. Besides this, the results are very similar to the results of previous experiments. Thus, the performance degradation faced by user CPU-bound process is still viewable in the latest version in the Linux kernel (2.6.24 as of May 2008).

In this section, we have studied the scope of the performance degradation issue faced by CPU-bound user processes in a networked Linux environment. The performance was assessed under different kinds of network I/O-bound processes, different types of network interface cards, and different versions of the Linux kernel. In all the cases, user CPU-bound processes were found to suffer from sever performance degradation around a network arrival rate of 90 Kpps. Thus, the performance issue is neither specific to the type of the network I/O-bound process, nor to the type of NIC used. In addition, the performance issue does exist in the latest kernel of the Linux OS (2.6.24 as of May 2008).

CHAPTER 4

ANALYSIS OF CPU-BOUND PROCESSES STARVATION UNDER NETWORK ENVIRONMENT

4.1 Introduction

In the previous chapter, the performance problem was assessed using different types of network I/O-bound processes and different network adapters (NIC's). The results showed that the problem does exist under all these configurations. Thus, the problem is neither specific to the type of the network I/O-bound process, nor to the type of NIC used. In addition, we studied the problem under two versions of the Linux kernel, namely, kernel 2.6.16 and kernel 2.6.24 (most recent). The problem was observed in both of the two kennels. Therefore, the assessment results indicate that this performance issue is most likely related to the operating system kernel, particularly, the process scheduler.

In order to make an in-depth analysis of the performance bottleneck at hand, we have to thoroughly study the Linux scheduling mechanism. The Linux scheduler has been greatly modified in the last few years. In fact, one of the major changes in the kernel of Linux 2.6, released to the public in December 2003, was an entirely new scheduling algorithm, mostly advertised for its O(1) complexity [LKA]. Later, in October 2007, the O(1) scheduler has been totally replaced with another scheduling

algorithm, the Completely Fair Scheduler (CFS) bundled with kernel 2.6.23 [LKA]. This newly introduced scheduling mechanism has been going through a sequence of major modifications and revisions since the time it was firstly released in 2007 [MOL07].

As the problem was detected under two totally different versions of the Linux scheduler, the problem should be separately analyzed under each of the two versions. In this chapter, we are going to analyze the performance problem in both versions of the Linux scheduler, namely, the old O(1) scheduler, and the newly introduced CFS scheduler. For the old version of the scheduler, we are going to study the performance of the scheduler in Linux kernel 2.6.16, while Linux kernel 2.6.24 is used to analyze the performance issue found in the new scheduler. It should be noted that the assessment as well as the analysis of the performance problem in Chapter 3 and Chapter 4 only handles uni-processor operating systems. The assessment and analysis of the problem under symmetric multiprocessing (SMP) environment will be discussed in Chapter 6.

The remainder of this chapter is organized as follows: Section 4.1 introduces a detailed analysis of the performance problem found in Linux O(1) scheduler (as of kernel 2.6.16). Then, the performance issue is thoroughly studied under Linux new CFS Scheduler (as of kernel 2.6.24).

4.2 Performance Analysis under the Linux 2.6 O(1) Scheduler.

The kernel of Linux 2.6 came with an entirely new scheduling algorithm, which was designed to achieve the following new features:

- A bound of O(1) on the time needed to choose the next process to run.
- Quick response to interactive processes under high system load.
- An acceptable level of prevention of both starving and hogging.
- Scalability and task affinity under symmetric multiprocessing environment.
- Improved performance when having a small number of processes [TOR07], [LOV05].

The following subsection details the basic algorithm design and specifications of the Linux kernel 2.6 O(1) scheduler.

4.2.1 Scheduler Basic Algorithm Design and Specifications

In general, the main data structure used by the Linux scheduler is the runqueue, which holds a list of all runnable tasks assigned to a particular CPU in the system. In a symmetric multiprocessing operating system, there is one runqueue per processor. The runqueue is divided internally into two arrays: the active array and the expired array (Figure 4.1). The active array contains all processes that are ready to run. On the other hand, the expired array contains all processes that are temporarily disabled as they have consumed their entire timeslices. When the active array becomes empty, it is switched with the expired array, which becomes active again. [TOR07], [LOV05]. This active-expired array design is credited for the O(1) performance of the scheduler [CRA07].

The active array consists of 140 lists corresponding to 140 possible priorities in the system. The first 100 priorities are reserved for real time processes. Processes are added to a particular priority list from its tail, and are consumed from the list's head. The

scheduler selects the next process to run from the head of the highest non-empty priority list. As the Linux 2.6 kernel is preemptible, when an interrupt occurs or the scheduler clock ticks, the running process will be preempted under the condition that it holds no kernel locks and a higher-priority task becomes available [ROD05], [KRO06].

The priority of a normal, also called conventional, process — as opposed to a real time process, which is out of the scope of this research context — consists of a sum of two values: a static priority called nice value, and a dynamic priority called bonus value. The nice value is a static priority value that indicates the process inherent relative importance. The nice value is, by default, inherited from the process's parent. It can be also changed by the process owner using the *nice()* and *setpriority()* system calls [BOV05]. There are 40 nice values corresponding to the priority levels from 100 to 139, as the priority levels from 0 to 99 are reserved for real time processes. Those nice values range in increasing order of priority from +19 (which corresponds to the priority level of 139) to -20 (which corresponds to the priority level of 100), with 0 as the default nice value (Table 4.1). The dynamic priority bonus value ranges from -5 to +5. It is dynamically assigned by the processor based on the interactivity of the process [TOR07], [LOV05].

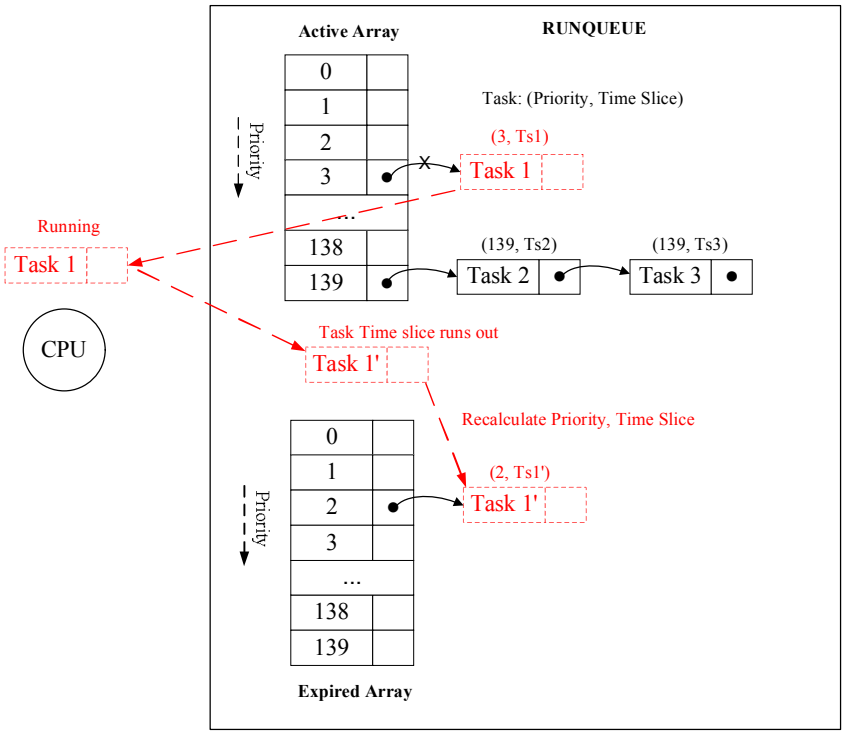

Figure 4.1: Linux kernel 2.6.16 scheduler runqueue and process scheduling [CRA07]

Each process is assigned a timeslice, also called time quantum, based on its nice value. Thus, higher priority processes are granted longer timeslices **(Table 4.1)**. The formula that is used to calculate the timeslice from the static priority is [BOV05]:

$$timeslice\ (ms) = \begin{cases} (140 - static\ priority) \times 20 & where\ static\ priotiy < 120 \\ (140 - static\ priority) \times 5 & where\ static\ priotiy \geq 120 \end{cases} \quad (4\text{-}1)$$

A process might not use its entire timeslice at once because it gets blocked for an I/O or preempted by a higher priority process. However, the process will eventually consume its entire timeslice, and then the process is placed in the expired array with a new timeslice and a recalculated priority [TOR07], [LOV05].

Table 4.1: Nice values and their corresponding timeslice lengthes [LOV05]

Task Priority Level	Nice Value	Timeslice Length
Initially created (parent's)	parent's	half of parent's
Minimum Priority (100)	+19	5ms (MIN_TIMESLICE)
Default Priority (120)	0	100ms (DEF_TIMESLICE)
Maximum Priority (139)	-20	800ms (MAX_TIMESLICE)

The scheduling of interactive tasks differs from the way described above. Although interactive tasks receive the same timeslice assigned to their peers of the same priority, their timeslices are subdivided into smaller pieces. When a subdivision of the timeslice is consumed, the process will round robin with other processes of the same priority. Thus, the execution is rotated more frequently among interactive tasks. Furthermore, when an entire timeslice of an interactive process is consumed, the process is not placed in the expired state. Rather, the process is given a new timeslice in the active array, unless processes in the expired array are starving or is having a higher-priority process. Starvation, in this case, occurs when the first expired process is older than some time limit, which is the amount of time proportional to the number of processes in the active array since the last switch between the active and expired arrays [TOR07], [LOV05]. In this case, processes that consume their entire timeslices are inserted into the expired array regardless of their interactivity level [TOR07], [LOV05].

As was mentioned earlier, the interactivity of a process is used to calculate its bonus priority value. In general, it is difficult for a low-priority process to qualify as an interactive process, while it is difficult for a high-priority process not to qualify as an interactive process. The interactivity of a process is measured based on its sleep average, a value that indicates how long the process has been sleeping. The value is increased

when the process is waiting and decreased when the process is running [TOR07], [LOV05]. In the following subsection, we focus more on the mechanism by which the dynamic priority and the sleep time of a process is calculated in the Linux 2.6 scheduler.

4.2.2 Dynamic Priority Calculation

The dynamic priority is the portion of the process priority that allows Linux kernel to adaptively change its scheduling behavior based on the process dynamic behavior. The dynamic priority value, the bonus value, is calculated using the process's average sleep time. Linux kernel keeps a track of a calculated value of average sleep time for each process using a long integer variable in the process structure called *sleep_avg*. The *sleep_avg* holds the calculated average sleep time in nanoseconds throughout the lifespan of a process. However, the calculation of the average sleep time of a process is not a simple average operation of elapsed time durations. Rather, its calculation is affected by process states and it is decremented as the process is running. Thus, *sleep_avg* does not really reflect the real value of the average sleep time of a process. *sleep_avg* is bounded between 0 ms and the scheduler parameter *MAX_SLEEP_AVG*, which is usually set to 1000 ms [BOV05].

On the other hand, the bonus value of each process is used with the static priority (i. e. the nice value) to calculate the process effective priority at any point in time. There are two different quantities that are usually confused when talking about bonus value, namely, the bonus value and the dynamic priority bonus. The bonus value is a value in

the range from 0 to the scheduler parameter *MAX_BONUS*, which is usually set to 10. It is calculated from the *sleep_avg* of the process using the following formula [CRA07]:

$$bonus\ (P,t) = \frac{P \to sleep\_avg \times MAX\_BONUS}{MAX\_SLEEP\_AVG} \quad (4\text{-}2)$$

The dynamic priority bonus, on the other hand, is calculated from the bonus value using the following formula [CRA07]:

$$dynamic\_priority\_bonus\ (P,t) = 5 - bonus\,(P,t) \quad (4\text{-}3)$$

Thus, the dynamic priority bonus is bounded between -5 (which corresponds to the highest bonus value, i.e. 10) and +5 (which corresponds to the lowest bonus value, i.e. 0). The process dynamic priority at any point in time is calculated from the static priority value (nice value) and the dynamic priority bonus using the following formula [BOV05]:

$$dynamic\_priority\,(P,t) = \max\{100, \min\{static\_priotiry(P) + dynamic\_priority\_bonus, 139\}\} \quad (4\text{-}4)$$

From (4-2), (4-3), and (4-4), it is obvious that Linux 2.6 Scheduler credits interactive processes, which have high *sleep_avg*, by increasing their priority and, on the other hand, penalizes non-interactive processes, which has low *sleep_avg*, by decreasing their priorities. Any value of bonus greater than 5 results in an increase in the process's effective priority, while any value of bonus below 5 results in a decrease in the process's effective priority [BOV05]. Table 4.2 lists the average sleep value ranges and their corresponding bonus and dynamic priority bonus values [BOV05].

Table 4.2: Process average sleep value ranges and their corresponding bonus and dynamic priority bonus values [BOV05]

Average Sleep Value Range (ms)	Bonus Value	Dynamic Priority Bonus
$0 \leq sleep\_avg < 100$	0	+5
$100 \leq sleep\_avg < 200$	1	+4
$200 \leq sleep\_avg < 300$	2	+3
$300 \leq sleep\_avg < 400$	3	+2
$400 \leq sleep\_avg < 500$	4	+1
$500 \leq sleep\_avg < 600$	5	0
$600 \leq sleep\_avg < 700$	6	-1
$700 \leq sleep\_avg < 800$	7	-2
$800 \leq sleep\_avg < 900$	8	-3
$900 \leq sleep\_avg < 1000$	9	-4
$sleep\_avg = 1000$ (*MAX_SLEEP_AVG*)	10	-5

Linux 2.6 Scheduler needs to differentiate between interactive and non-interactive processes, as the interactivity of a process affects the way it gets handled by the scheduler. The interactivity of a process is determined dynamically using the process's average sleep time value, *sleep_avg*, along with the process's static priority. The criterion that the process has to meet in order to qualify as an interactive process is expressed by the following formula [BOV05]:

$$dynamic\_priority \leq \frac{3}{4} \times static\_priotiry - 28 \qquad (4\text{-}5)$$

Formula (4-5) can be expressed in terms of bonus value as follows [BOV05]:

$$bonus - 5 \geq \frac{1}{4} \times static\_priority - 28 \qquad (4\text{-}6)$$

More precisely,

$$bonus \geq \frac{1}{4} \times static\_priority - 23 \qquad (4\text{-}7)$$

The right hand side of formula (4-6) is called *interactive delta*. A closer look at formulae (4-6) and (4-7) shows that it is much easier for a high priority process to qualify as interactive, while it is much more difficult for a low priority process. On the extreme, a process of lowest priority, i. e. 139, would never be classified as interactive, as this requires the bonus to be 11, to reach an interactive delta of 6. As the bonus calculations is based on average sleep time of the process, i. e. *sleep_avg*, an interactivity qualification threshold of *sleep_avg* can be calculated for each priority level. For the default priority level, i. e. 120, the threshold value for *sleep_avg* is 700 ms. Thus, if a process of priority level 120 has a sleep average greater than or equal to 700 ms, it is classified as an interactive process. Table 4.3 shows some of these threshold values [BOV05].

Table 4.3: Interactive delta values and sleep time thresholds for several priority levels [BOV05]

Description	Static Priority	Interactive Delta	Sleep Time Threshold
Highest static priority	100	-3	200 ms
High static priority	110	-1	400 ms
Default static priority	120	2	700 ms
Low static priority	130	4	900ms
Lowest static priority	139	6	1100ms

4.2.3 Sleep Time Estimation

As was mentioned in the previous section, the average sleep time of a process is not a simple average function of the time the process has spent sleeping. Rather, its calculation is affected by process states and it is decremented as the process is running.

In addition, the average sleep time is also affected by the interactive status of the process. The average sleep time of a process is updated in one of the following two cases:

1. When a process yields the CPU.
2. When a process wakes up from sleep or blocking state.

The two cases are different in the way in which the average sleep time is calculated.

When a process yields the CPU, its average sleep time, *sleep_avg*, is calculated using the following formula [CRA07]:

$$P \to sleep\_avg_{now} = \max\left\{0, P \to sleep\_avg_{old} - running\_time \times \alpha\right\}, \qquad (4\text{-}8)$$

where

$P \to sleep\_avg_{now}$ is the newly calculated *sleep_avg*,

$P \to sleep\_avg_{old}$ is the old value of *sleep_avg*,

$running\_time$ is the amount of time the process was running (ns), and

α is a weighting factor for runtime, calculated using the following formula [CRA07]:

$$\alpha = \frac{1}{\max\left\{1, bonus(P, t_{exec})\right\}}, \qquad (4\text{-}9)$$

where t_{exec} is the most recent point in time when P resumed its execution on the CPU.

On the other hand, when a process wakes up from a sleep or blocking state, its average sleep time, *sleep_avg*, is calculated according to the following formula [CRA07]:

$$P \to sleep\_avg_{now} = \min\left\{ MAX\_SLEEP\_AVG, P \to sleep\_avg_{old} + sleep\_time \times \beta \right\}, \quad (4\text{-}10)$$

where

MAX_SLEEP_AVG is the maximum possible value for *sleep_avg*, configured to 1000 ms by default,

$P \to sleep\_avg_{old}$ is the old value of *sleep_avg*,

sleep_time is the amount of time the process was sleeping (ns), and

β is a weighting factor for sleep time, calculated using the following formula [CRA07]:

$$\beta = \max\left\{1, 10 - bonus(P, t_{sleep})\right\}, \quad (4\text{-}11)$$

where t_{sleep} is the point in time when P started sleeping.

Several scenarios are treated in a special way when updating the average sleep time of a process, *sleep_avg*. First of all, processes that sleep for a very long time are classified as idle; therefore, they are given minimally interactive status. This is necessary as to avoid the possibility that they suddenly become CPU hogs, and starving other processes. In addition, when a process wakes up from an uninterruptible sleep, the rise in its *sleep_avg* is limited, as it is most likely doing a disk I/O. Finally, the time processes spent in the runqueue after they wake up, the scheduling latency, which could be a considerable amount of time, is conditionally credited to the process's *sleep_avg*, depending on the state of the process when it got woken up. The state of the process when it was awakened is encoded in the process's *activated* field. Table 4.4 lists the possible values of the *activated* field, their meanings, and the time credited to *sleep_avg* from scheduling latency [CRA07], [BOV05].

Table 4.4: activated values, their meanings, and the percentage of scheduling latency credited to *sleep_avg* for each value [BOV05].

***activated* Value**	**Description**	**Percentage of Scheduling Latency Credited to *sleep_avg***
0	The process was in *TASK_RUNNING* state.	N/A
1	The process was in *TASK_INTERRUPTIBLE* or *TASK_STOPPED* state, and it is being awakened by a system call service routine or a kernel thread.	100%
2	The process was in *TASK_INTERRUPTIBLE* or *TASK_STOPPED* state, and it is being awakened by an interrupt handler or a deferrable function.	30%
-1	The process was in *TASK_UNINTERRUPTIBLE* state and it is being awakened.	0

The place in the kernel scheduler where most of these *sleep_avg* calculations and update are performed is the function *recalc_task_prio()*. This function performs two important tasks: *sleep_avg* estimation and update, and dynamic priority calculation. Thus, this function is very important in our analysis of the performance issue at hand. Figure 4.2 shows the control flow-chart of this function. [BOV05]

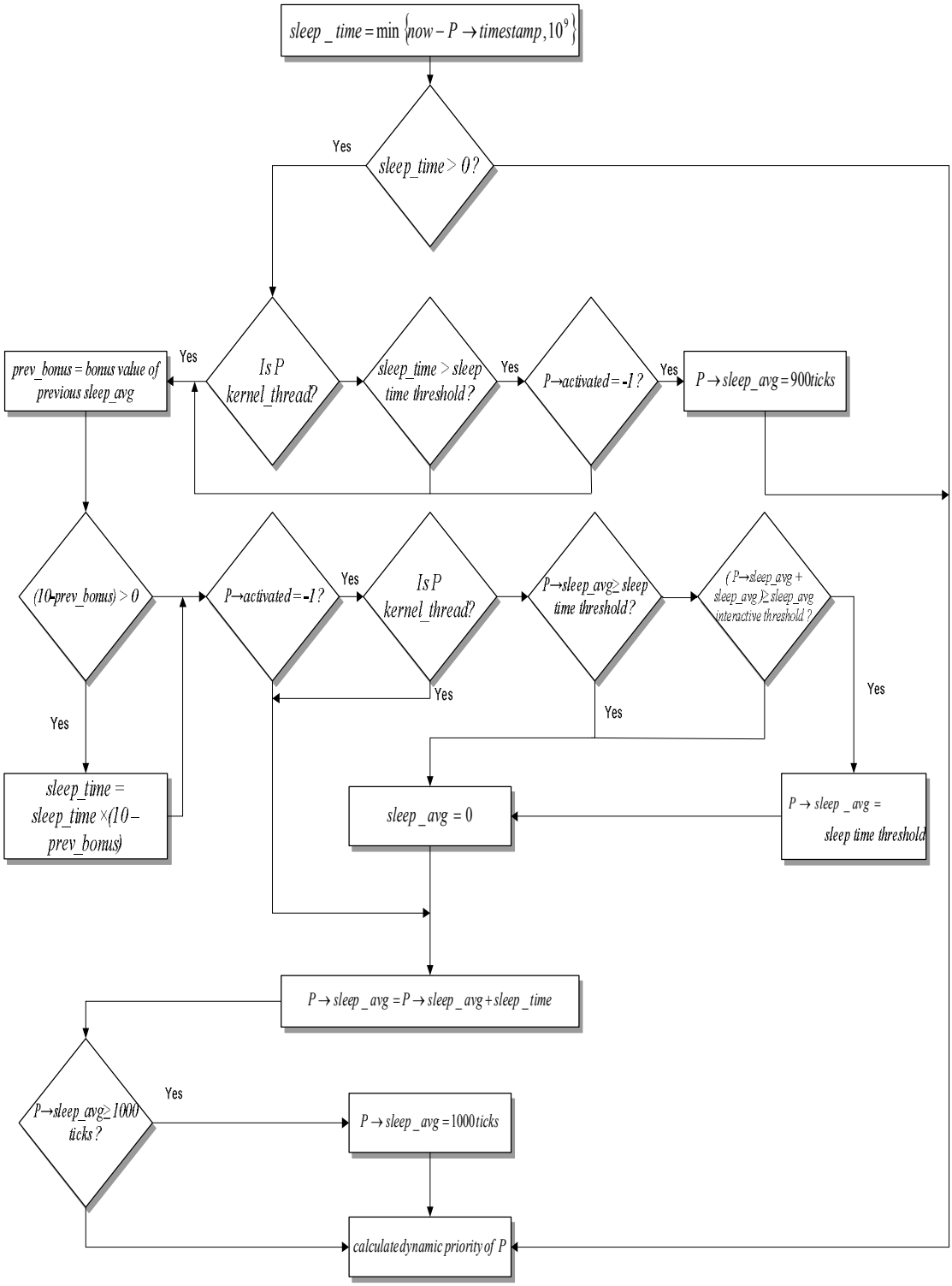

Figure 4.2: Control flow-chart of *recalc_task_prio()* function

4.2.4 Analysis of CPU-Bound Processes Starvation

Now, as we have built a comprehensive picture of the process scheduler in Linux 2.6 kernel, the scheduler of kernel 2.6.16 in particular, we would take a deeper look into the performance issue encountered by CPU-bound processes during certain network flow rate, i. e. 70-100 Kpps. It is apparent from the problem assessment results discussed in Chapter 3 that the problem issue is caused by some "misbehavior" of the process scheduler. The ideal behavior expected from the process scheduler is to give the CPU-bound processes a monotonically decreasing amount of CPU resources as the network load or flow increases. Figure 4.3 shows one possibility of the ideal behavior (linear trend) along with the real behavior.

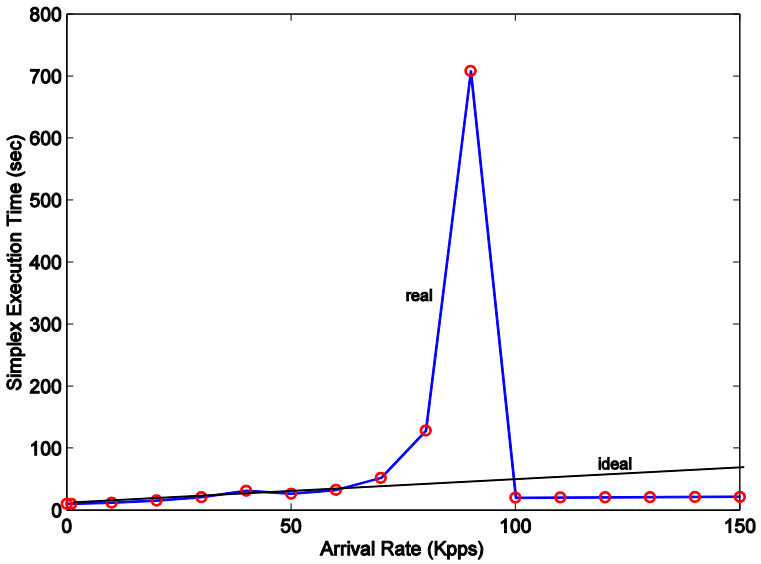

Figure 4.3: Ideal and real behavior of Linux 2.6 O(1) scheduler

From the basic background given in Section 4.1.3, we know that the basic criterion based on which the process scheduler favors one process over the other is the process dynamic priority. The dynamic priority of a process consists of two values, the

static priority (i. e. nice value) set by the user, and the bonus that is dynamically calculated using heuristics that take the value of the process's sleep time as the basic parameter for calculation. If the user of a process does not specify the nice value, the system sets that value to a unique default value. In our problem assessment, we never set the nice value of neither the network nor the Simplex processes. Thus, both network processes and Simplex process have the same default nice value, which is 0. From the above given facts we can make a conjecture that the source of this scheduler "misbehavior" is basically the bonus value, particularly the average sleep time value, calculation of the process. Put in different words, the problem is directly related to the dynamic priority calculation mechanism of the process scheduler, which is based on the estimation of the average sleep time of each process.

In order to verify our conjecture, we have to modify, test and analyze the behavior of the scheduler. Particularly, we have first to disable the process dynamic priority calculation mechanism in the scheduler and test its performance. If disabling the process scheduler dynamic priority mechanism causes the performance problem to disappear, then this proves our conjecture. The next step is to take different measurements for the calculated average sleep time of the network process, to analyze its changes and effects on the performance of the Simplex process. Figure 4.4 depicts in a flow-chart model the logic of the performance problem analysis.

With the above analysis requirements, we have to set up a testing environment, where we analyze the behavior of the scheduler using various kernel instrumentation techniques for both taking measurements and introducing changes and assessing their

effects on the performance of the scheduler. The following heading details the testing environment used to analyze the problem.

- **Experimental Setup.** In all of the experiments carried out for analyzing the performance problem of Linux O(1) scheduler, we use the same experimental setup shown in Figure 3.1. The recipient is loaded with Kernel 2.6.16. For simplicity, we have chosen Distributed Internet Traffic Generator (D-ITG) as the network flow generator (network processes *ITGSend* and *ITGRecv*) for all our testing cases, although choosing other network tools, like KUTE, would produce the same results as has been proved in Chapter 3. The CPU-bound user application is represented by the Simplex process, which was set to run for 1 second on a free CPU in all performance analysis experiments.

- **Dynamic-Priority-Disabled Scheduler.** To verify that the performance problem of the Linux 2.6 O(1) scheduler is directly related to the dynamic priority calculation of processes, we disabled the dynamic priority calculation of Linux Scheduler in kernel 2.6.16, and then tested the performance of this "dynamic-priority-disabled" Linux Scheduler. To disable the dynamic priority calculation, we modified the *recalc_task_prio()* function such that it does not do any updates to the average sleep value of any process, i. e. *sleep_avg*. We build the 2.6.16 kernel with this modified scheduler. See Appendix A for more details.

We tested this modified scheduler using D-ITG package and Simplex. As was inspected earlier in our conjecture, the results of the experiment show that the performance problem of CPU-bound processes, represented by Simplex in this

experiment, totally disappears under the "dynamic-priority-disabled" Linux O(1) scheduler. This proves that the performance issue at hand is directly related to the dynamic priority calculation mechanism of the scheduler. Figure 4.4 shows the results of this experiment.

Figure 4.4(a) shows the relation between the time Simplex takes to execute in seconds and the network traffic rate in Kpps. The graph shows a monotone increasing linear trend, although the linear slope is not unique throughout the traffic rate space from 0 to 150 Kpps. In particular, the trend is linear with a small positive slope from 0 to 50 Kpps. Then the slope sharply increases between 60 and 90 Kpps. From 100 and up to 150 Kpps, the slope decreases, having a value near its value between 0 and 50 Kpps. Thus, the trend is always increasing and there is no instability or sharp jumps in the trend. This clearly shows that disabling the dynamic priority calculation mechanism has caused the performance problem of the scheduler to disappear.

In addition, with a closer look at Figure 4.4(a), we notice that the delay encountered by Simplex due to the increase in the system network load is very small compared to the time Simplex spends in the original "dynamic-priority-enabled" scheduler, which can reach to more than 700 seconds at the sharp peak between 70 and 100 Kpps. In particular, at the greatest network load on the system, which is 150 Kpps in this experiment, the Simplex delay is less than 1.5 seconds. Thus, disabling the dynamic priority calculation module in the scheduler has stabilizes and improved the scheduler behavior.

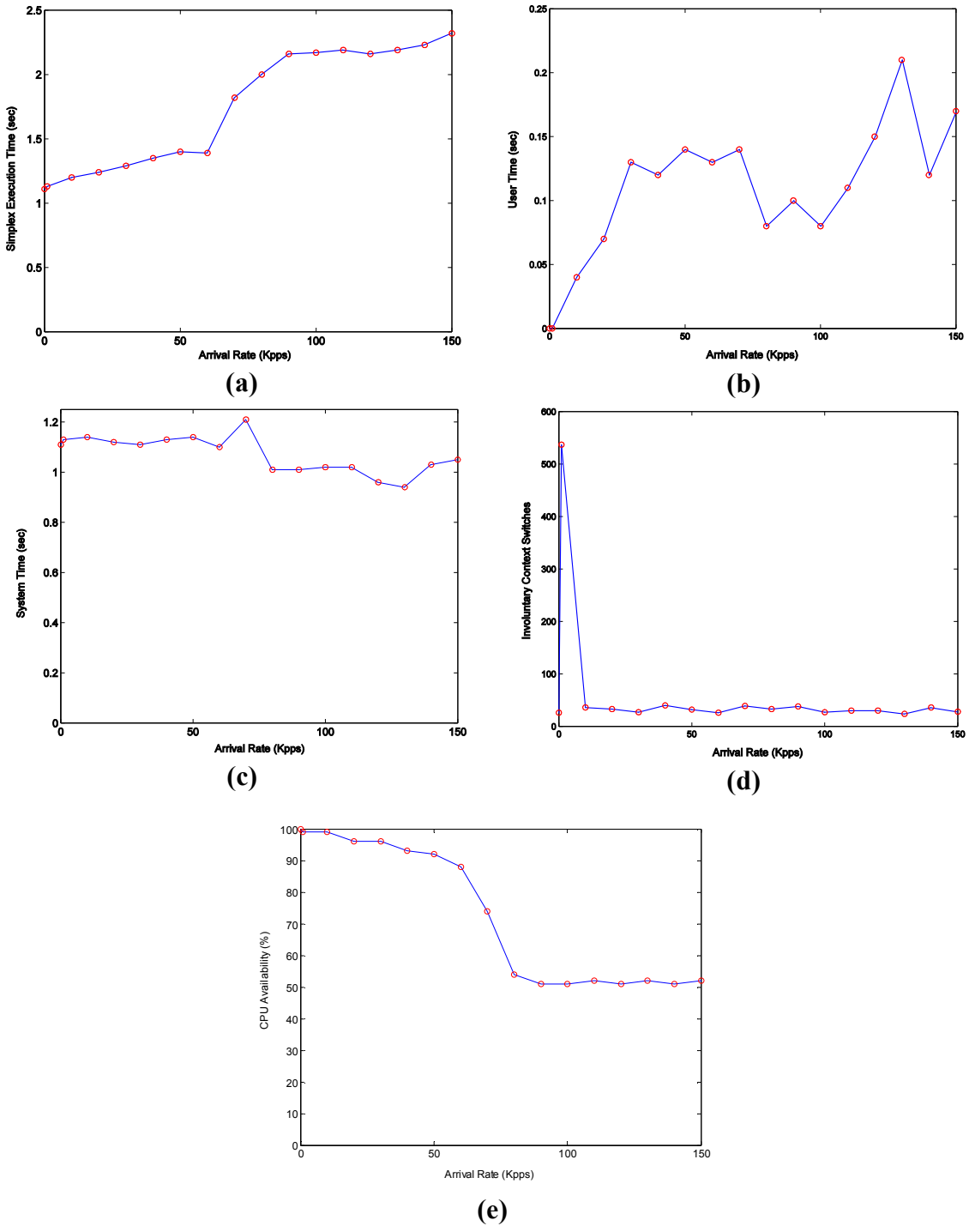

Figure 4.4: Performance measurement with "dynamic-priority-disabled"

Figure 4.4(b) and (c) show the system and user portions of the total Simplex time. The most noticeable thing in the two graphs is the small delay in both of them, as compared to the results we got in the original scheduler. The user time is less than 0.2 seconds at the highest network load, and the system time is less than 1.25 seconds at the highest peak. This small delay explains the fluctuations found in both trends, as the noise effect becomes more apparent in small scale, as opposed to large scale.

Figure 4.4(d) shows the involuntary context switches that Simplex is forced to by our "dynamic-priority-disabled" scheduler. Excluding the spike found at very low traffic rate (1000 Kpps in particular), the involuntary context switches number remains below 50. This small number illustrates the fact that the performance issue faced by CPU-bound processes is resolved when disabling the scheduler dynamic priority calculation module, as the Simplex process is not forced out of CPU too many times while its consuming its timeslice. Even the spike at 1000 Kpps is not very high (below 550) and could be considered as an outlier.

Figure 4.4(e) shows a clear monotonically decreasing trend in the amount of CPU resources available for Simplex process. In a perfect agreement with Figure 4.4(a), the decrease is gradual between 0 and 50 Kpps, and then it sharply steepens between 60 and 90 Kpps. From 100 Kpps and on, the amount of CPU resources available for Simplex stays almost constant around 50 percent. Thus, the results in Figure 4.4(e) support our first conjecture of accusing the dynamic priority calculation module of the scheduler of the observed performance instability problem.

In summary, the above results show that the dynamic priority calculation mechanism of the Linux 2.6 O(1) Scheduler is directly related to the performance

degradation issue of user CPU-bound processes in the network load range between 70 and 100 Kpps. Thus, the dynamic priority calculation mechanism, and the average sleep time calculations in particular, should be closely studied to find out the root cause of the observed performance instability in the scheduler.

As Simplex is, by definition, a CPU-bound user application; the way the Linux scheduler handles Simplex process should not be affected by disabling the dynamic priority calculation module in Linux scheduler. In fact, we have run Simplex on both the original "dynamic-priority-enabled", and the modified "dynamic-priority-disabled" Linux schedulers, and they spent precisely equal times. This is explained by the fact that Simplex process never have a "voluntary" type of sleep that Linux 2.6 O(1) scheduler uses as the base of dynamic priority calculations. Simplex process is always preempted and never yields the CPU unless it finishes all of its calculations, where it dies and yields the CPU. Thus, enabling or disabling the dynamic priority calculation through controlling the amount of average sleep time (i. e. *sleep_avg*) granted to each process on the system has no effect Simplex, as its average sleep time is always zero. On the other hand, the network process has frequent "voluntary" sleeps, as it basically does a network I/O, which requires it to sleep frequently waiting for packets delivery by the hardware driver. Thus, the network process should be treated differently when disabling the dynamic priority calculation in the scheduler. Therefore, we need further analysis of how the dynamic priority of the network process is handled in the Linux 2.6 O(1) scheduler. The following section contains an analysis of how the network process scheduling parameters are handled in the Linux 2.6 O(1) scheduler.

- **Analysis of the Scheduling Parameters of the Network Process (*ITGRecv*).** To understand how the scheduler handles the dynamic priority of the network process, we introduced an instrumentation code in the Linux kernel such that we can trace the sleep time values of the network process (*ITGRecv* in this case) during different network loads. In addition, we also keep track of other related scheduling values including the total number of sleeps, the average sleep time value *sleep_avg*, the dynamic priority, and the interactivity status of the network process. The collection process is triggered and terminated externally from the user space. The instrumentation code was introduced to the scheduler code without affecting its original control flow; see Appendix B for more details.

Using this instrumentation in the kernel, we run a simple experiment similar to the one in Figure 3.1. The recipient machine is loaded with the kernel that has the instrumentation code. We send network traffic from the sender to the recipient for a specific period of 30 seconds, during which the scheduling variables are traced and reported. We vary the network traffic rate from 1 Kpps to 120 Kpps and collect the scheduling variables values at each step. The results we got are shown in Table 4.5, and Figure 4.5 and 4.6. In the following, we discuss the results in more detail.

Table 4.5 and Figure 4.5 show a random sample of the *sleep_time* values of the network process at the critical network traffic rate (i. e. 85 Kpps in this experiment). Those results clearly show the basic sleeping behavior of the network process at the critical network traffic rate. In most cases, the network process (*ITGRecv*) sleeps for a

duration of time that is less than 10 milliseconds. In fact, the sleep time was below 10 milliseconds in more than 88% of the cases.

Table 4.5: *ITGRecv sleep_time* values at 85 Kpps taken from a random sample of more than 500 readings

sleep time (ms)	Frequency	Percentage of Sample	Cumulative Frequency	Percentage of Sample
0	0	0.0%	0	0.0%
4	231	40.2%	231	40.2%
8	278	48.3%	509	88.5%
12	33	5.7%	542	94.3%
100	9	1.6%	551	95.8%
104	12	2.1%	563	97.9%
200	6	1.0%	569	99.0%
208	6	1.0%	575	100.0%

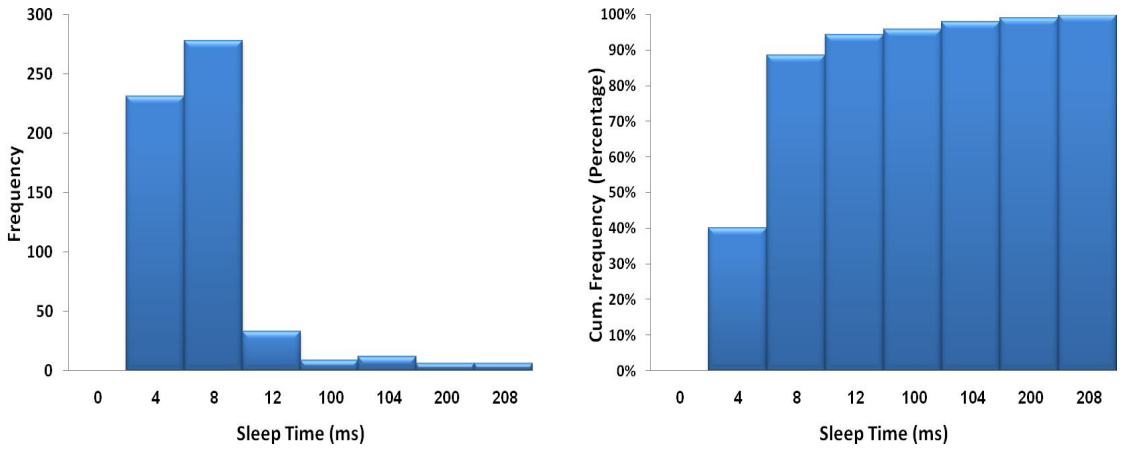

Figure 4.5: *ITGRecv* sleep_time values at 85 Kpps taken from a random sample of more than 500 readings

In Figure 4.6(a), the average length of a every single sleep of the network process is plotted against varying network traffic rate. As the figure clearly shows, the average length of a single sleep of the network process is about 4 ms at all traffic rates less than 85 Kpps. After 85 Kpps, the network process have almost no sleeps, thus the

value of the length of a single sleep cannot be defined at this stage. Thus, we have conveniently set the value to zero in this traffic range.

On the other hand, Figure 4.6(b) shows the total number of sleeps that *ITGRecv* has made per second at different network traffic rates. The figure shows an interesting behavior that can be interpreted by referring to the concept of Linux NAPI, which is a packet reception mechanism implemented in Linux 2.6 to alleviate the problem of receive livelock through "interrupt coalescing". In Linux NAPI, when the "first" packet arrives and copied to the device DMA ring, an interrupt (i. e. RxInt) is raised to notify the CPU about the availability of the packet, the network device is added to the poll list, and all further packet arrival interrupts are disabled. Thus, new arriving packets are being copied to the device DMA silently. When the network process wakes up and processes all the packets in the DMA, the interrupts are re-enabled again.

At low network traffic load, i. e. between 0 Kpps and 40 Kpps, the number of sleeps per second is actually linear, such that there is almost one sleep per one arriving packet. This is because the packets arrival rate is very slow compared to the network process packet processing rate, such that one packet is completely processed and removed from the device DMA before the next packet actually arrives. Thus, NAPI re-enables the RxInt interrupts and the network process goes to sleep waiting for the next packet. This is why we have almost one sleep per one arriving packet. However, beyond 40 Kpps, the rate of packet arrival exceeds the rate of the network packet processing rate, such that new packets arrive while the network process is still processing some previously arrived packets. The newly arrived packets are copied silently to the device DMA, and the network process handles them before going to sleep. This is why we see a

sharp drop in the number of sleeps after 40 Kpps. However, at this traffic range, the packet arrival rate is not too high, such that the network process will eventually consume all packets in the DMA before its timeslice expired, and thus go to sleep again. This situation causes a slight increase, or mostly a balance, in the number of sleeps of the network process between 40 Kpps and 70 Kpps. Eventually, the network process timeslice would not suffice to process all the packets in the DMA, thus its timeslice is totally consumed and the process is preempted. This is why we see another sharp drop after the 85 Kpps, where the network process has almost no sleeps at all.

Figure 4.6(c) shows the average value of the calculated dynamic priority of the network process at different traffic rates. From very low traffic rates and up to 80 Kpps, the network process dynamic priority is at its maximum possible value, i. e. 115, as the network process is assigned the default nice value, namely 120. However, after 80 Kpps the dynamic priority starts dropping until it reaches 118 at almost 87 Kpps. This drop in the dynamic priority of the network process is due to the decrease in the number of sleeps the process makes at this high traffic rate, which would result in a decrease in its *sleep_avg* value, and eventually its dynamic priority. This explains why Simplex performance greatly improves beyond this particular network traffic rate. However, the dynamic priority does not drop to the minimum possible value, namely 125. This can be understood from the analysis of the *sleep_avg* value of the network process presented in Figure 4.6(d).

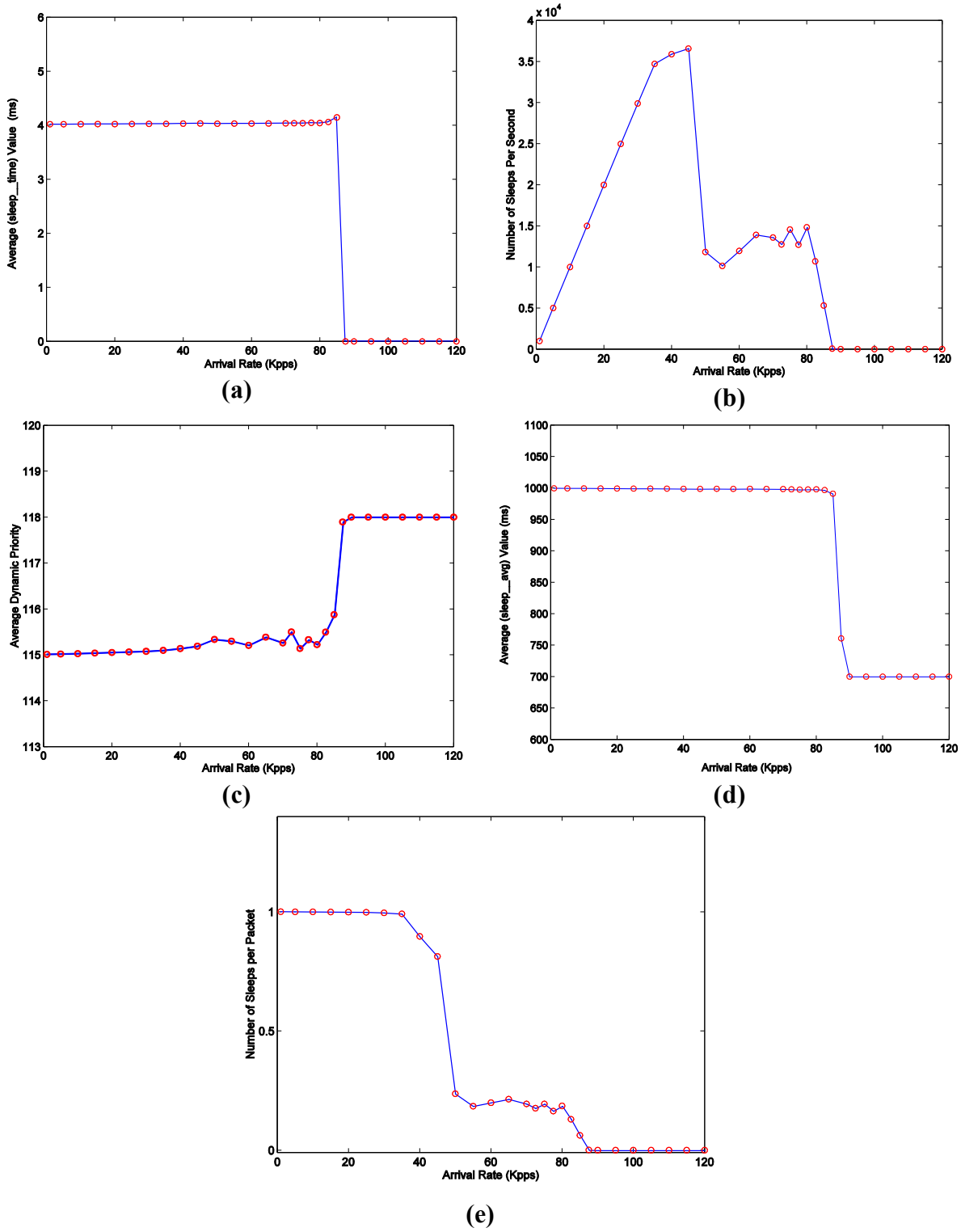

Figure 4.6: *ITGRecv* sleep behavior analysis

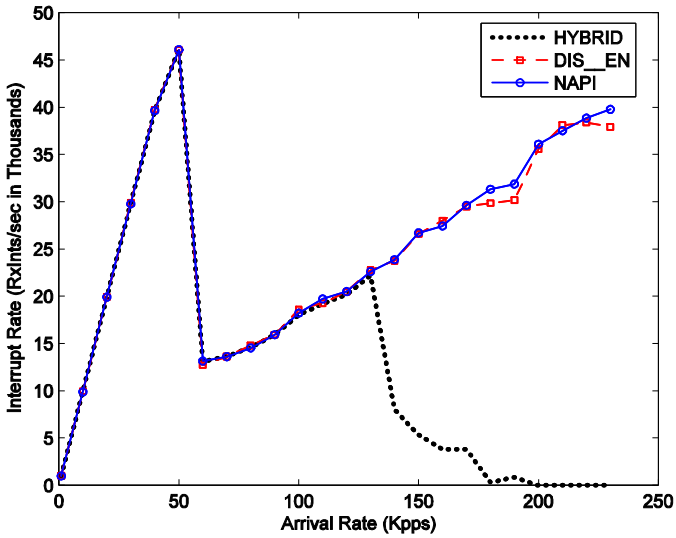

Figure 4.7: Interrupt rate for NAPI and other interrupt handling schemes at different traffic loads [QAH07]

In Figure 4.6(d), the average value of the *sleep_avg* of the network process is plotted against the network traffic arrival rate. It is apparent from the figure that *sleep_avg* is at its maximum value (i. e. *MAX_SLEEP_AVG,* which is set to 1000 ms by default). This explains why the network process's dynamic priority is boosted to the maximum possible value of 115 in the range from 0 Kpps to 80 Kpps. This is caused by the large number of short network sleeps (around 4 ms) that accumulate in *sleep_avg*, boosting its value to the maximum possible value of 1000 ms. As the number of sleeps reduces beyond 85 Kpps barrier, the value of *sleep_avg* drops to 700ms level and stays stable there. The reason why *sleep_avg* does not drop further is actually the way in which the scheduler handles interactive tasks. When a task is classified by the scheduler as an interactive task, the penalty the scheduler charge on the task's *sleep_avg* as the task is running is proportionally reduced based on the task's previous bonus value. This makes it less probable for interactive tasks to lose their interactivity as they are running.

Thus, even though the number of sleeps has reduced, the *sleep_avg* never drops below 700 ms, which is the interactivity threshold for tasks with the default nice value of 0. We went further and analyzed the interactivity status of the network process at all traffic rates. Our results have shown that the network process is classified as an interactive process for all traffic rates, which agrees with the results in Figure 4.6(d).

Figure 4.6(e) shows the number of sleeps the network process makes for every arriving packet. The trend is similar to the trend in Figure 4.6(b), where we have one sleep per packet at low traffic rates, particularly less than 40 Kpps. Then, the rate sharply decreases after 40 Kpps, as more packets are being serviced in one CPU-burst of the network process. However, between 40 Kpps and 70 Kpps, the packet arrival rate is not too high, such that the network process will eventually consume all packets in the DMA before its timeslice expired, and thus go to sleep again. This happens more frequently, as the arrival rate is increasing, thus causing a slight increase in the number of sleeps the network process makes. This explains the balance in the rate of sleeps per packets in this traffic range. Eventually, the network process timeslice would not suffice to process all the packets. Thus, it is totally consumed and the network process is preempted. This is why we see another sharp drop in the rate of sleeps per packet beyond 85 Kpps, where the rate becomes almost zero at 90 Kpps.

To analyze the situation even further, we have made another instrumentation to collect the number of timeslice consumptions and expirations for the network process (*ITGRecv*) at different traffic rates, as shown in Figure 4.8. Timeslice consumptions give an overview of the amount of CPU resources the network process gets at different traffic rates. On the other hand, network process expirations give an idea of how the scheduler

handles the network process interactivity and the system starvation at different traffic rates. Basically, only non-interactive process are expired in 2.6 O(1), unless the tasks in the expired array are deemed starving. This happens if any of the following two conditions holds:

1. Some task in the expired array has a better static priority than the interactive task.
2. The time since the first process in the active array expired is greater than or equal to the following value:

$$STARVATION\_LIMIT \times nr\_running + 1 , \qquad (4\text{-}12)$$

where

STARVATION_LIMIT is a pre-defined starvation limit, which is set by default to the value of *MAX_SLEEP_AVG*, which has a default value of 1000 ms, and

nr_running is the number of processes in the run queue.

Figure 4.8(a) shows the number of timeslice consumptions of *ITGRecv* in two different situations, with and without Simplex being run in the background. In general, the number of timeslice consumptions increases as the traffic rate increases, in a good agreement with the results shown in Figure 4.6. In particular, timeslice consumptions increases linearly between 0 Kpps and 40 Kpps. This is logical, as the increase in the network traffic rate will cause the network process to take more processing time, and thus consumes more timeslices. Then, the rate of increase in timeslice consumptions temporarily drops between 40 Kpps and 60 Kpps. This is because at this traffic rate, the network process starts processing more packets in a single CPU burst, which increases the processing efficiency, as it reduces the overhead of context switching and cache

pollution. After that, the rate of increase in timeslice consumption sharply increases, until 85 Kpps, where it reaches a maximum value of almost 300 consumptions (without Simplex running) and 150 consumptions (with Simplex running). This is expected as the increase in the traffic rate causes the network to consume more timeslices to handle the increasing traffic. Beyond 85 Kpps, the number of timeslice consumptions remains constant as traffic rates increases. This is actually maximum throughput that the system can achieve.

On the other hand, Figure 4.8(b) shows the number of network process expirations with increasing traffic rate. As our earlier analysis of Figure 4.6 has shown, the network process is always classified as interactive, and thus, the only cause of the network process expirations is the starvation conditions shown earlier. We see that there almost no expirations between 0 and 85 Kpps. This means that the network process is always in the active array between 0 and 85 Kpps. After the 85 Kpps barrier, the number of expirations increases jumps to almost 300 expirations (without Simplex running) and 150 consumptions (with Simplex running).

The result in Figure 4.8 gives a good insight into the starvation issue. First, comparing the results of the experiments were Simplex was running with those were Simplex was not running show very clearly how CPU-bound processes can starve in the presence of network I/O bound processes. From 0 Kpps to 85 Kpps, whether we run Simplex in the background or not makes no difference in the obtained results. This means that the network process monopolizes the CPU, such that the Simplex is only able to run when the network process yields the CPU. Thus, it is not surprising that when the network process consumes more timeslices, and hence CPU resources, Simplex process

starts to starve. However, when the network process priority is decreased, beyond 85 Kpps, we see clearly the difference between the two cases. When Simplex is running, the network process's consumptions of timeslices is almost half (i. e. 150) of that when Simplex is not running (i. e. 300). This clearly shows that at this traffic rate, the CPU resources is divided fairly between Simplex and the network process. This is why we see a boost in the performance of Simplex beyond 85 Kpps.

In addition, comparing the results in Figure 4.8(a) and 4.8(b) can give another good insight into how the scheduler handles the network process at different traffic rates. First, between 0 and 85 Kpps, non of the timeslice consumptions of the network process causes it to expire. Thus, the network process, though consuming many timeslices, is deemed interactive and is given a large share of the CPU resources. The situation is worst just before 85 Kpps, when the network process has a large number of timeslice consumption (and hence a large share of CPU resource), and yet it is never being penalized for that. This allows the network process to monopolize the CPU resources and causes other process in the system to starve. However, beyond 85 Kpps, the scheduler heuristics for detecting starvation in the expired array holds, and thus we see that regardless of the interactivity status of the network process, it is being expired whenever its timeslice is totally consumed. Thus, we see that the number of timeslice consumptions and expirations is almost the same beyond 85 Kpps.

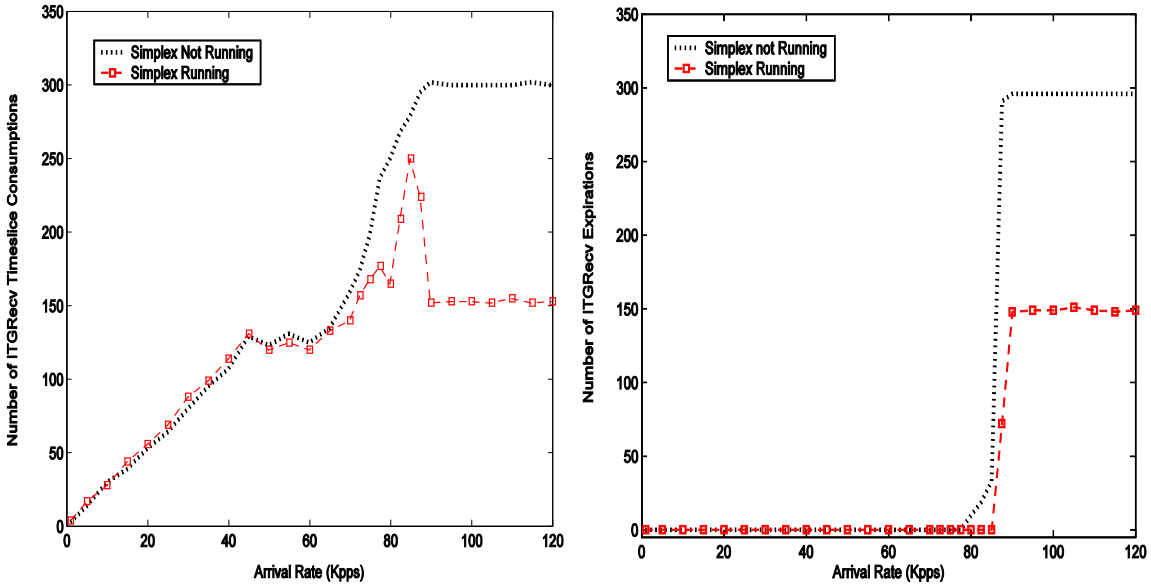

Figure 4.8: *ITGRecv* timeslice consumptions and expirations

Figure 4.9 shows the number of times the network process (*ITGRecv*) was reinserted into the active array of the runqueue (after its timeslice has been totally consumed) while varying the network load. As the 2.6 O(1) scheduler reinserts only interactive process into the active array after their timeslices have been totally consumed, this value, which we call, for simplicity, *reinsertion count*, is a strong indicator of the process interactivity. Nonetheless, the 2.6 O(1) scheduler has starvation heuristics, which helps avoiding starvation of processes in the expired array because of continuous reinsertion of interactive processes in the system. Thus, as those starvation limits holds, the scheduler stops reinsertion, and all process that consumes their entire timeslice are moved to the expired array, regardless of their interactivity status. Thus, the reinsertion count can also indicate whether the interactive processes in the system are causing starvation to the processes in the expired array or not.

In Figure 4.9, the reinsertion count is shown in two cases: with and without simplex being run in the background. In both cases, the reinsertion count increases as the network traffic increases from 0 to 85 Kpps. However, beyond 85 Kpps, the reinsertion count drops to almost 0, and never changes while the network traffic increases. The increase in the reinsertion count of the network process from 0 to 85 Kpps clearly indicates that the network process is classified as interactive over that period. On the other hand, there could be two causes of the drop in the reinsertion count beyond 85 Kpps: either the network process loses its interactivity status, or the scheduler has stopped reinsertion to prevent starvation. As the network process *ITGRecv* is scheduled with the default priority, the values of *sleep_avg* of the network process in Figure 4.6(d) shows that it is always classified as interactive, since its *sleep_avg* value is always above 700 ms (see Section 4.2.2 for more details on process interactivity calculations). Thus, the drop in the reinsertion count is really caused by the starvation limits that the scheduler enforces.

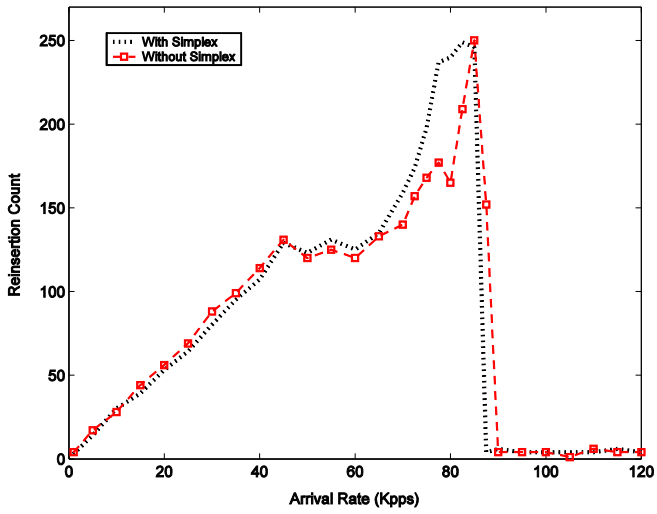

Figure 4.9: Reinsertion count of *ITGRecv* at different network loads

In general, the reinsertion count increases as the traffic rate increases, in a good agreement with the results shown in Figure 4.8. However, the rate of the increase in reinsertion count between 0 and 85 Kpps is not uniform. In particular, between 0 and 50 Kpps, the rate is almost constant, giving the two curves a linear shape. This is logical, as the increase in the network traffic rate will cause the network process to take more processing time, and thus consumes more timeslices. Then, the rate of increase in reinsertion count temporarily drops between 40 Kpps and 60 Kpps. This is because at this traffic rate, the network process starts processing more packets in a single CPU burst, which increases the processing efficiency, as it reduces the overhead of context switching and cache pollution. After that, the rate of increase in reinsertion count sharply increases, until 85 Kpps, where it reaches a maximum value of almost 250 reinsertions .This is expected as the increase in the traffic rate causes the network to consume more timeslices to handle the increasing traffic.

Figure 4.9 shows that the presence of CPU-bound processes in the system has no effect on the interactivity status of the network process, as the two curves coincide over each other. This means that the network process monopolizes the CPU, such that the Simplex is only able to run when the network process yields the CPU. Thus, it is not surprising that when the network process consumes more timeslices, and hence CPU resources, Simplex process starts to starve. In fact, misclassifying the non-interactive network processes as interactive gives them an extra “unfair” share of the CPU resources on the expense of other CPU-bound processes. This misclassification, at certain traffic loads (85 Kpps in this experiment), allows the network process to monopolize CPU recourses and cause CPU-bound processes to starve, despite the presence of the 2.6 O(1)

scheduler starvation heurists, which are unable to recognize the starvation of CPU-bound processes at this specific traffic load.

As was detailed in Section 4.1.3, the *sleep_time* value of any process has an upper limit, such that if its value exceeds that limit, it is not totally granted to the process's *sleep_avg*. Those "long-sleeping" processes are classified as idle, and the scheduler is designed to limit their dynamic priority level (and, in effect, their interactivity status) such that they do not monopolize the CPU at any point in time. However, the scheduler design does not have any lower limit for the duration of time a process may sleep. Thus, all processes that sleep for a small amount of time are given full advantage of their sleep time, unless other different conditions limit their sleep time evaluation.

This scheduler design, which does not impose any kind of lower limits on the process's sleep time value to be counted in its average sleep time, is prone to a potential risk. A long sleeping batch process that could suddenly become a CPU hog is supposed to be demoted by the process scheduler such that it does not take over the system's CPU resources and cause other processes to starve. Although the scheduler is designed to recognize such risky processes, it is only able to recognize them if they tend to sleep for a very long time. However, if their "long-sleeping" time is divided into many "short-sleeping" periods, the scheduler can never point them out. On the contrary, the scheduler promotes such processes as it misclassifies them as interactive processes, and gives them more CPU resources. The network process (*ITGRecv* in this case) is an example of those processes, which the scheduler is unable to estimate their dynamic priority level and interactivity correctly.

The network receiving process is usually non-interactive and is supposed to leave CPU processing power for other tasks in order to prevent starvation of other processes in the system. However, as the results show, the network receiving process (*ITGRecv*) have around 90% of its sleep time below 10 ms, and thus it gets full advantage of its sleep time value, affecting the performance of other CPU-bound user processes (Simplex process in this experiment). In fact, Wu and Crawford have shown in their theoretical and experimental work in [CRA07] that "the 'relatively fast' non-interactive network process might frequently sleep to wait for packet arrival. Though each sleep lasts for a very short period of time, the wait-for-packet sleeps occur so frequently that they lead to interactive status for the process."

In summary, the analysis of the *sleep_time* value of the network process (*ITGRecv*) has shown that the sleeping behavior of the network process can cause the scheduler to boost its dynamic priority and treat it as an interactive process. This is because the scheduler design lacks the ability to recognize non-interactive processes that have many “short-term” sleeps. In order to ensure that this limitation in the scheduler design is the root cause for the observed instability in the performance of CPU-bound processes, we have to filter out the network process’s “short-sleeps” and study how this affects the performance problem. This analysis is discussed in detail in the following section.

- **Limiting Sleep Time (*sleep_time*) Values of the Network Process (*ITGRecv*).** In order to clarify the effect of the lack of lower limits on the value of sleep time (*sleep_time*) in the Linux 2.6 O(1) scheduler design and the “short-sleeps” behavior of

the network process (*ITGRecv* in this case) on the instability in the performance of CPU-bound use applications, we modified the Linux 2.6 scheduler code such that it includes an imposed static lower limit on the value of the sleep time (*sleep_time*) of any process. The lower limit (*MIN_SLEEP_TIME_MS*) was defined to be the lower limit for the *sleep_time* value in milliseconds. Figure 4.10 shows a portion of the modification code inserted into the Linux 2.6 O(1) scheduler code. It is important to notice that the goal at this point of the research is to clearly identify the root cause of the performance problem, while we focus on the complete solution in Chapter 5.

In order to study the effect of the introduced lower limit on the behavior of the scheduler, we varied the *MIN_SLEEP_TIME_MS* from 15 ms all the way to 1000 ms, which is the maximum value for the average sleep time of a process *sleep_avg*, i. e. *MAX_SLEEP_AVG*. For each lower limit value, the performance of the CPU-bound user application (Simplex) was logged using the *time* utility. The results of the experiments are shown in Figure 4.10. They include the Simplex time, user time, system time, involuntary context switches and CPU available resources for every tested value of the process's sleep time lower limit *MIN_SLEEP_TIME_MS*. For the sake of comparing the modified scheduler results with the original results, we also tested the performance of Simplex while setting the lower threshold value to 0 ms, which is basically equivalent to the original scheduler logic that does not impose any limitation on the minimum value of the sleep time of any process.

```
...
...

static int recalc_task_prio(task_t *p,
unsigned long long now)
{
        ...
        ...



        p->sleep_avg += sleep_time;


        if (p->sleep_avg >
NS_MAX_SLEEP_AVG)
                p->sleep_avg =
NS_MAX_SLEEP_AVG;
        return effective_prio(p);
}
```

Original Scheduler Code

```
/* we vary this lower limit */
#define MIN_SLEEP_TIME_MS       10
static unsigned long sleep_ms = 0

...
...

static int recalc_task_prio(task_t *p,
unsigned long long now)
{
        ...
        ...
        sleep_ms = sleep_time/1000000;
        if (sleep_ms > MIN_SLEEP_TIME_MS)
        {
                p->sleep_avg +=
sleep_time;
        }

        if (p->sleep_avg >
NS_MAX_SLEEP_AVG)
                p->sleep_avg =
NS_MAX_SLEEP_AVG;
        return effective_prio(p);
}
```

Modified Scheduler Code

Figure 4.10: Original vs. modified scheduler code

Figure 4.11(a) clearly shows the effect of introducing a limiting threshold on the minimum allowed value of the sleep time of the network process. In a perfect agreement with the previous *sleep_time* screening experiment results, which shows that almost 90% of its values are below 10 ms, the results in Figure 4.11(a) show how introducing a lower limit of just 15 ms extremely reduces Simplex execution time at the two peak (40 Kpps and 75 Kpps). At the larger peak (75 Kpps), the saved time is actually more than 170 seconds. On the other hand, the threshold value does not have a major impact on Simplex execution time before and after the two peaks, which clearly shows that introducing the lower threshold helps stabilizing the performance of Simplex. In addition, increasing the lower limit value beyond 15 ms (from 30 ms to 1000 ms) has a very small impact on the Simplex execution time. This agrees with the *sleep_time*

screening experiment results, as only 10% of the network process's sleep time values are more than 10 ms.

Figures 4.11(b), (c) and (d), show similar trends to the trend in Figure 4.11(a). In Figure 4.11(b), the user time portion of Simplex execution time is graphed with changing lower limit values, and it shows the large impact of introducing a 15-ms-threashold limiting the value of *sleep_time*. Figures 4.11(c), also shows the same great impact on the system time portion of the Simplex execution time. However, the reduction in system time is much more than that in user time, as most of the Simplex long execution time at the peaks is comprised of system time (scheduling, context switching, etc). The reduction in the number of involuntary context switches Simplex was forced to is shown in Figure 4.11(d). In particular, introducing a lower limit of 15 ms reduces the number of involuntary context switches by more than 2.25 million context switches at the large peak (75 Kpps). Finally, all the figures show that increasing the lower limit value beyond 15 ms gives a minor improvement effect, as expected by the results of the previous *sleep_time* sampling experiment.

Figure 4.11(e) shows the effect of introducing different lower threshold values limiting the *sleep_time* value on the amount of CPU resources available to Simplex. The results show that the lower threshold values allowed Simplex to get more CPU resources, as the graphs move upwards (which means more CPU resource percentages) by increasing the threshold value. For the threshold values of 15ms, 30ms, and 100ms, the CPU resources available to Simplex does not change between network rates of 0 Kpps and 30 Kpps. However, the difference appears beyond 30 Kpps, where it is clear that the more the threshold value, the more CPU resources Simplex can get. The results

also the change in CPU resources available to Simplex diminishes beyond network rate of 80 Kpps, where the graphs merge again. This actually agrees with the results in Figure 4.11(a), (b), (c) and (d), where the only largely spotted performance improvement is between network rates of 30 Kpps and 80 Kpps.

However, the results in Figure 4.11(e) differs from the other results in Figure 4.11 in the fact that the amount of CPU resources available for Simplex continues to increase by increasing the threshold value beyond 15 ms. This is explained by the fact that introducing a lower threshold value of 15 ms helps the scheduler in blocking the network process from being an interactive process, and thus removes the overhead on the scheduler by scheduling the high-priority network process which sleeps too many times for short periods and wakes up immediately to ask for the CPU and cause an involuntary context switch. Thus, this threshold value ideally blocks the network process and saves the scheduling overhead, which has a clear effect on the performance of Simplex, as it solves the root cause of its performance degradation problem, i. e. the unjustifiable classification of network process as interactive. However, increasing the value of the threshold beyond 15 ms starts to affect other processes in the system, as their sleep times are not being counted in their sleep averages. This, of course, gives Simplex an unfair advantage of more CPU resources on the expense of other processes in the system. On the extreme, when setting the threshold value to 1000 ms, which means that almost no process in the system can be interactive, we see that Simplex monopolizes the CPU resources, as it always gets more than 70% of the CPU resources available in the system.

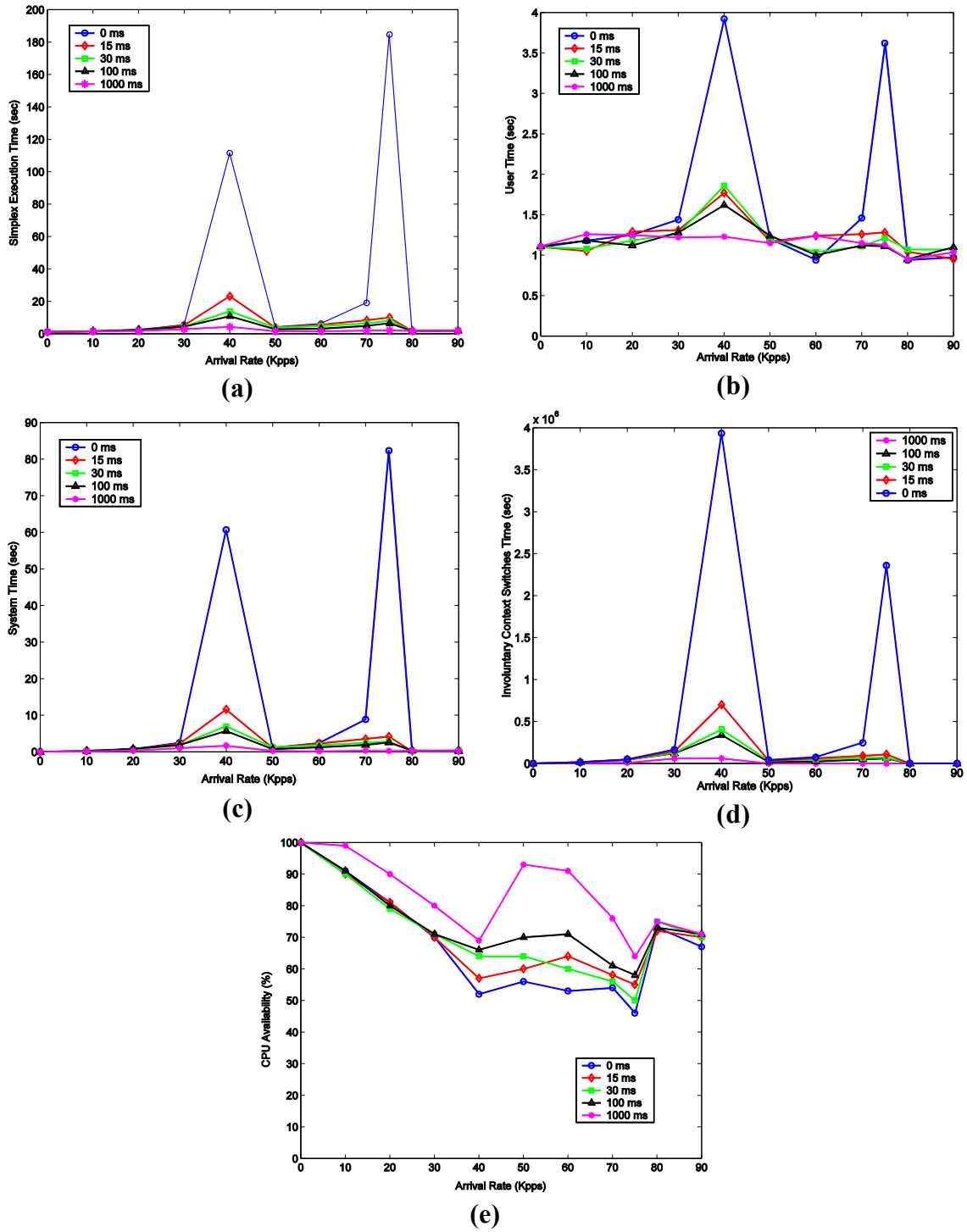

Figure 4.11: Results of varying the minimum sleep time threshold from 5 ms to 1000 ms

In summary, the results we got from limiting the sleep time value (*sleep_time*) of the network process (*ITGRecv*) using a global lower threshold value show a great performance improvement in the CPU-bound processes (*Simplex*). The performance improvement is clearly viewable as soon as the threshold is set beyond 10 ms limit. No great gain in performance is noticed when increasing the threshold value from 15 ms all the way to 1000 ms. Rather, the CPU-bound processes might get unfair share of the CPU resources by increasing the amount of the threshold beyond 15 ms.

The focus in this section was to clearly identify the root cause of the performance instability problem encountered by CPU-bound processes in a networked Linux system running 2.6 O(1) scheduler. The root cause of the problem was found to be the inability of the scheduler to correctly classify the network process as a non-interactive process. This is because the scheduler design lacks the ability to recognize the "frequent-short-sleeping" behavior of the network process, which allows it to get full advantage of all of its non-interactive sleeps, and thus gets a boosted priority. We proved this experimentally by modifying the scheduler to have a global threshold value on the minimum sleep time value that is allowed to be counted in the average sleep time of any process. The modified scheduler outperforms the original scheduler, and has more stable performance. The complete solution of the performance problem is discussed in detail in the next chapter (Chapter 5).

4.3 Performance Analysis under Linux CFS

In Linux kernel 2.6.23, the O(1) scheduler has been completely replaced with a new scheduling algorithm, named as the Completely Fair Scheduler (CFS). The new scheduler is a complete rewrite of the Linux task scheduler carried out by Ingo Molnar, who based his work on the fair scheduling work of Con Kolivas; mainly his implementation of the Rotating Staircase Deadline Scheduler (RSDS). The CFS was designed to maximize CPU utilization and the system's interactive performance by fixing the deficiencies found in the previous O(1) vanilla scheduler. One of the main features of CFS, besides its "fair scheduling" algorithm, is its modular design, where processes are scheduled according to scheduling classes. Another important difference between the old and the new scheduler is that the new CFS has no notion of a specific pre-set timeslice per process. Rather, the timeslice is totally dynamic and its calculation is implicitly accomplished by the CFS algorithm [MOL07].

The scheduler has not fully stabilized yet, since it is only two years old and is continuously being reviewed, updated, and modified. In this study, we have based our work on the latest version of the Linux CFS available at the current time of the research, which is the CFS in kernel 2.6.24 released to the public on January 2008 [LTL]. In the following subsection, the basic design features of CFS are explained in more detail.

4.3.1 CFS Basic Algorithm and Features

The basic idea behind the Linux new CFS is to emulate an ideal and precise multitasking uni-processor CPU, which is able to run all the processes in the system in parallel and with an equal speed of *1/n*, where *n* is the number of runnable tasks in the system. As the real uni-processor system hardware is able to run only one task at any point in time, the CFS algorithm strives to make an equal and fair division of the CPU bandwidth among all tasks in the system. This fair multitasking is accomplished by tracking the time each runnable task spends waiting for the CPU. In the context of CFS, when a task runs on the CPU, the other runnable tasks are unfairly waiting for the CPU. Thus, the time each waiting task has to run on the CPU to become fair with other runnable tasks is recorded in a specific per-process field originally called *wait_runtime*, as of kernel 2.6.23. This quantity has been changed in kernel 2.6.24 to a different quantity called *vruntime*, which holds a weighted sum of the execution time of every process [MOLN07], [KUM08], [WON08].

Thus, and as understood by the definition of the *wait_runtime* quantity, the *wait_runtime* value of a task is incremented while the task is waiting for the CPU, and is decremented when the task is actually running. The value by which the *wait_runtime* is incremented or decremented is determined by the number of runnable tasks in the system at any point in time. This was originally (i. e. in kernel 2.6.23) managed by a runqueue global clock called *fair_clock*, which runs at a certain pace that is fractional to the real time pace. This clock has been changed in kernel 2.6.24 to a local per-process clock equal to the ratio of the wall clock to the task weight. Thus, as soon as a waiting

task has its *wait_runtime* value greater than the currently running task (with some additional latency that is used to avoid over-scheduling), it preempts the running tasks. Thus, *wait_runtime* in kernel 2.6.23, or *vruntime* in kernel 2.6.24, is the quantity based on which CFS determines what task to run next and whether a running task should be preempted or not. It is important to notice that *wait_runtime* and *vruntime* calculations take into account the static priority of the process [KUM08], [WON08].

After introducing the basic idea of the Linux CFS, we give more details about the new scheduler's runqueue with the scheduling classes and entities, and then we focus on how the *vruntime* calculations are performed.

- **CFS Main Runqueue, Scheduling Classes, and Scheduling Entities.** As discussed in this section, the Linux CFS schedules processes according to their *vruntime* value. Thus, it should use a data structure that allows for fast storing and fetching of tasks according to their *vruntime*-sorted values. To meet such a requirement, CFS uses a time-ordered red-black binary tree for every processor in the system [MOL07]. The red-black tree is a special type of the binary search tree, which is self-balancing and encodes every node with either red color or black color. It has special constraints including that the root and leaves must be black, and that the children of every red node must be black. Those special constraints make sure that the worst-case time complexity for insertion or deletion of nodes is O(log n), where n is the number of nodes in the tree. Each runnable process, or scheduling entity according to CFS terminology, in the system is encoded by an internal node in the tree. The leaf nodes in a red-black tree must be left empty and are not allowed to contain data; however, they are used to simplify the operation algorithms

on the tree. Thus, they are sometimes implemented as a single sentinel node, in order to save memory [RBT]. In the red-black tree of CFS in kernel 2.6.23, the processes are inserted to the tree according to the key "*rq->fair_clock - se->wait_runtime*", where *rq* is the runqueue and *se* is the scheduling entity (i. e. the task). In kernel 2.6.24, the key used in inserting element to the tree is the value "*se->vruntiem – cfs->min_vruntime*". In either way, the process at the left-most node in the tree is always the one most eligible to run next on the CPU [MOLN07], [KUM08], [WON08].

CFS creates a runqueue for each for each processor in the system. The runqueue contains a list of the runnable entities on a given processor. In every runqueue, there is CFS-related field (*struct cfs_rq*) and real time related field (*struct rt_rq*). The CFS-related field is a data structure that holds runnable schedulable entities in the system. Scheduling entities can be, in hierarchical order, containers, groups, users, or tasks. These different kinds of scheduling entities are used to implement CFS different fair scheduling schemes, including user-fair scheduling and group-fair scheduling. If user-fair scheduling and group-fair scheduling are disabled, the scheduling entity is equivalent to a task in the system. As scheduling entities are hierarchical, a scheduling entity might be enclosing other scheduling entities. Thus, when the scheduler selects a scheduling entity that should run next, it first checks if this scheduling entity is a runnable task. If it is not a runnable task (e. g. container entity), it continues the lookup for all scheduling entities enclosed within the runqueue of the top level entity. This process continues down the hierarchy, until a runnable scheduling entity (i. e. task) is found [WON08].

Unlike Linux 2.6 O(1) scheduler, which uses different scheduling policies to schedule different kinds of tasks, Linux CFS uses a modular framework, which defines scheduling classes as an extensible hierarchy of standalone scheduling modules that encapsulate scheduling policy details to schedule different types of tasks. The scheduling core handles those scheduling modules without care about the modules fine details. The old *SCHED_OTHER* scheduling policy, which was used to schedule desktop normal time-sharing tasks, is replaced in CFS by the *sched_fair* scheduling module. In addition, real time tasks are scheduled using the *sched_rt* scheduling module, which implements the old *SCHED_FIFO* and *SCHED_RR* scheduling policies [WON08], [MOL07].

This subsection gives some details about the basic algorithm and features of the Linux newly introduced CFS. The focus on the following subsection is shifted towards the calculations of the basic quantity that CFS uses to sort tasks competing for CPU resources in the system, namely, the *vruntime* quantity.

- **Calculation of the Task's *vruntime* (Kernel 2.6.24).** As was highlighted in this section, *vruntime* is the value based on which CFS make decisions on how to schedule different tasks in the system. Each task in the system has its own *vruntime*, which represent a weighted sum of the execution time of the task. More precisely, *vruntime* is defined by the following formula [KUM08], [WON08]:

$$se \rightarrow vruntime_{n+1} = se \rightarrow vruntime_{n} + \frac{se \rightarrow delta\_exec}{se \rightarrow load.weight} \times NICE\_0\_LOAD \quad , \qquad (4\text{-}13)$$

where

se is the scheduling entity,

$se \rightarrow vruntime_{n+1}$ is the new value of *vruntime*,

$se \rightarrow vruntime_{n}$ is the old value of *vruntime*,

$se \rightarrow delta\_exec$ is the amount of the execution time of the task,

$se \rightarrow load.weight$ is the weight of the task, mapped from its nice value according to the *prio_to_wight[ ]* array, and

NICE_0_LOAD is the unity value of the weight, i. e. the mapped weight value to the nice level 0, which is 1024

In order to give a better explanation of *vruntime*, we have to talk about another important scheduling variable introduced in kernel 2.6.24, namely, the scheduling period (*sched_period*). The Linux CFS tries to fairly schedule every task once in the scheduling period, *sched_period*, which is set o 20 ms by default. However, the fair shares of the period that all ready tasks in the system get are not equal. Rather, those shares are stretched or shrunk according to the task's weight. The formula used to calculate the task's slice of the scheduling period at any point in time is as follows: [KUM08], [WON08]:

$$se \rightarrow slice = \frac{se \rightarrow load.weight}{cfs\_rq \rightarrow load.weight} \times sched\_period \ , \qquad (4\text{-}14)$$

where

$se \rightarrow slice$ is the slice of the scheduling period,

$se \rightarrow load.weight$ is the weight of the task, mapped from its nice value according to the *prio_to_wight[ ]* array,

$cfs\_rq \rightarrow load.weight$ is the sum of the weights all tasks in the CFS runqueue, and

$sched\_period$ is the scheduling period.

The scheduling period is not constant. Rather, it can be set through the *sysctl* parameter *sysctl_sched_latency*. In addition, when the number of running tasks in the system, i. e. *nr_running*, is greater than the parameter *sched_nr_latency*, the scheduling period, *sched_period*, get extended according to the formula [WON08]:

$$sched\_period = \frac{nr\_running}{sched\_nr\_latency} \times sysctl\_sched\_latency \ , \qquad (4\text{-}15)$$

where

nr_running is the number of running tasks in the system,

$sysctl\_sched\_latency$ is the initial value of the scheduling period, and

$sched\_nr\_latency$ is the ratio of the following two tunable parameters: *sysctl_sched_latency* and *sysctl_sched_min_granularity* according to the following formula [LKA]:

$$sched\_nr\_latency = \frac{sysctl\_sched\_latency}{sysctl\_sched\_min\_granularity} \ , \qquad (4\text{-}16)$$

where *sysctl_sched_min_granularity* is a tunable parameter that is used to define the minimal preemption granularity for CPU-bound tasks. [LKA].

By substitution from formula (4-14) in (4-12), another formula for calculating *vruntime* is [WON08]:

$$se \rightarrow vruntime_{n+1} = se \rightarrow vruntime_{n} + \frac{sched\_period}{cfs\_rq \rightarrow load.weight} \times NICE\_0\_LOAD \quad (4\text{-}17)$$

There are several tuning parameters of Linux CFS, which can be set through the *sysctl* command. Besides the *sysctl_sched_min_granularity* and the *sysctl_sched_latency* introduced earlier, there is the parameter *sysctl_sched_child_runs_first* which is used to decide which one of the parent process and its forked child should run first. In addition, other important parameters are the *sysctl_sched_batch_wakeup_granularity* and the *sysctl_sched_wakeup_granularity* which are used for *SCHED_BATCH* and *SCHED_OTHER* scheduling policies, respectively. Those parameters control the preemption effect of decoupled workloads, and reduce over-scheduling [LKA].

4.3.2 Analysis of CPU-Bound Processes Starvation

After giving an introduction to the basic features of Linux CFS as of kernel 2.6.24, we focus on the performance issue encountered by CPU-bound processes during certain network flow rate, i. e. 70-100 KPPS. The problem assessment results, discussed in Chapter 3, clearly shows that the problem is directly related to the process scheduler, as the ideal behavior expected from the process scheduler is to give the CPU-bound processes a monotonically decreasing amount of CPU resources as the network load or flow increases. Figure 4.12 shows one possibility of the ideal behavior (linear trend) along with the real behavior.

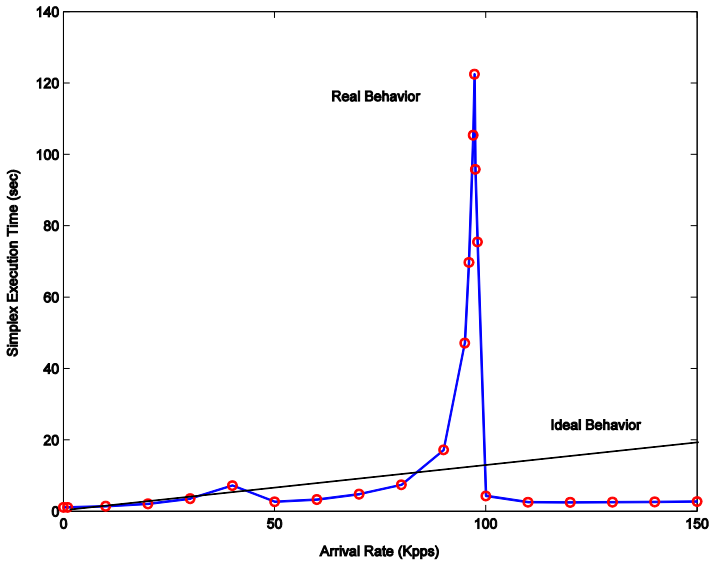

Figure 4.12: Ideal and Real Behavior of Linux CFS

When analyzing the issue in Linux 2.6 O(1) Scheduler, it was found that the root cause of the problem is the special sleeping behavior of the network process, along with the disability of the scheduler to recognize such a threat. As the scheduler is, by definition, a system module that manages the CPU resources among several competing processes without affecting the inherent behavior of any process, it is logical when analyzing the performance issue under Linux CFS to relook at the network process. For simplicity, we will assume that the sleeping behavior of the network process does not change by changing the system scheduler, which is, logically speaking, a reasonable assumption. Thus, it is very probable that performance problem under Linux CFS is directly related to the sleeping behavior of the network process.

However, unlike Linux 2.6 O(1) scheduler, which uses the sleep time of a process as the basic parameter for calculating the process's dynamic priority, Linux CFS focuses more on the running time of each task. Thus, profiling the sleep time of the network process is not an easy task in Linux CFS. Nonetheless, CFS comes with many

tunable parameters that allows for an immediate real-time interaction with the system scheduler. Those tunable parameters are a very good starting point for analyzing the performance problem under Linux CFS.

One of the tunable parameters in Linux CFS, which is directly related to the sleeping behavior of any normal process (i. e. processes that are scheduled under the *SCHED_OTHER* scheduling policy), is the parameter *sysctl_sched_wakeup_granularity* introduced earlier in this section. This parameter controls the wake-up granularity of *SCHED_OTHER* decoupled workloads to delay their preemption effect and reduce over-scheduling. In kernel 2.6.24, this parameter is initialized by the scheduler according to the following formula:

$$sysctl\_sched\_wakeup\_granularity = 10ms \times (1 + \log(ncpus)) \ , \qquad (4\text{-}18)$$

where *ncpus* is the number of CPU's in the system. Thus, the default value of this parameter is 10ms in single-processor systems. It should be noted that the default value of this parameter has changed in kernel 2.6.27 to 5ms [LKA].

For normal tasks, which include the network process *ITGRecv* and Simplex, the scheduler checks whether a "just-waking" task should preempt the current task or not. This checking is done by the function *check_preempt_wakeup* defined under *sched_fair.c* scheduler module. After doing several checking for processes belonging to different scheduling policies including real-time, batch and group scheduling, this function compares the values of *vruntime* for the current process and the newly waking-up process. If the value of *vruntime* of the current process is greater than the value of the *vruntime* of the newly waking-up process plus the wake-up granularity threshold (i. e.

sysctl_sched_wakeup_granularity), the waking-up process preempts the current process. Figure 4.13 shows a simplified version of the function *check_preempt_wakeup* [LKA].

```
static void check_preempt_wakeup(struct rq *rq, struct task_struct *p)
{
         struct task_struct *curr = rq->curr;
         struct cfs_rq *cfs_rq = task_cfs_rq(curr);
         struct sched_entity *se = &curr->se, *pse = &p->se;
         unsigned long gran;

              .
              .
              .

         gran = sysctl_sched_wakeup_granularity;

              .
              .
              .

         if (pse->vruntime + gran < se->vruntime)
                 resched_task(curr);
}
```

Figure 4.13: A simplified version of the check_preempt_wakeup funtion

The experimental analysis of the network process *ITGRecv* sleep behavior, performed under the Linux 2.6 O(1) Scheduler, has shown that the network process tends to frequently sleep for very short durations. Thus, and assuming that the network process behavior does not change under Linux CFS, it is expected that increasing the value of *sysctl_sched_wakeup_granularity* will reduce the number of times *ITGRecv* can preempt Simplex, which would, in turn, improve the performance of Simplex.

To experimentally prove this conjecture, we set up an experiment similar to the one shown in Figure 3.1. The sender is running Linux 2.6.16, and the recipient, is running Linux 2.6.24. The sender sends network traffic with rate 95 Kpps, using *ITGSend* utility. As was previously explored in Chapter 3, at this traffic rate the performance of Simplex is the worst. The recipient system is receiving the traffic using

ITGRecv utility, while running Simplex at the same time. The value of the parameter *sysctl_sched_wakeup_granularity* is varied in the recipient side while the performance of Simplex is recorded. The command used to vary the value of the parameter *sysctl_sched_wakeup_granularity* is:

```
# sysctl -w kernel.sysctl_sched_wakeup_granularity="10000000"
```

The command assigns the value of 10 ms to the parameter, as the parameter value is measured by nanoseconds [LKA].

We varied the value of *sysctl_sched_wakeup_granularity* from 0.1 ms to 30 ms, passing by the default value 10 ms. The results of this experiment are shown in Figure 4.14. As anticipated earlier, the general trend in the graphs shows that Simplex performance degrades as the value of the parameter decreases, and improves when the parameter value increases.

In Figure 4.14(a), the Simplex execution time is plotted against the varying value of the wake-up granularity parameter. In general, as the value of the wake-up granularity parameter increases, Simplex execution time decreases, and, thus, the performance of Simplex improves. However, with a close look at the plot, we can divide the trend it follows into three distinguishable regions: from 1 ms to 10 ms, from 10 ms to 20 ms, and from 20 ms to 30 ms. For simplicity, we call these regions, small, medium, and large wake-up granularity regions, respectively. Those three different regions are discussed in detail below.

- **Small Wake-up Granularity Region.** Although Simplex execution time decreases, the rate at which it decreases is very slow, as if it were constant. In fact, at 1 ms, Simplex execution time is 112 s, and at 10 ms, the execution time becomes 109 s,

reducing only 3 s. This observation can be directly mapped to the general sleeping behavior of the network process (Table 4.5**,** Figure 4.5), which shows that more than 80% of the network process (*ITGRecv*) sleeps are shorter than 10 ms. To understand the relation between the observed behavior and the network process sleep behavior, we give a simple example. Assume we have a system that only runs Simplex and *ITGRecv* (this is actually the case in this experiment, ignoring all system and terminal processes). In addition, assume that both Simplex and *ITGRecv* start together at the same time, and have the same nice value, or load weights, according to CFS terminology. Linux CFS algorithm strives to give an equal share of CPU bandwidth to both processes. Thus, at the end of any scheduling period (*sched_period*), CFS would, by the definition of CFS, have given both processes equal amounts of CPU time. Therefore, at the beginning of the next scheduling period, both Simplex and the network process will have almost equal values of *vruntime*. Let us analyze both Simplex and the network process at the beginning of any scheduling period. Both processes would have an almost equal amount of *vruntime*, call it x. Assume the network process starts the execution for a small duration of time, call it ε. Thus, its *vruntime* value becomes $x + \varepsilon$. Then, Simplex starts execution, while the network process is sleeping. Assume the network process sleeps for some time, say y. When the network process wakes up, Simplex would have executed for almost y ms. Thus, its *vruntime* value becomes $x + y$. At this point, the Linux CFS has to make a decision on whether the network process should preempt Simplex or not. This is done in the *check_preempt_wakeup* function according to the following formula:

$$vruntime_{Network} + sysctl\_sched\_wakeup\_granularity < vruntime_{Simplex} \quad (4\text{-}19)$$

If the above condition holds, Simplex would be preempted by the network process. By substituting the values in the above example in Formula (4-19) we get:

$$x + \varepsilon + sysctl\_sched\_wakeup\_granularity < x + y \quad (4\text{-}20)$$

Simplifying both sides of Formula (4-20), we get:

$$\varepsilon + sysctl\_sched\_wakeup\_granularity < y \quad (4\text{-}21)$$

Formula (4-21) can be rewritten as:

$$sysctl\_sched\_wakeup\_granularity < y - \varepsilon \quad (4\text{-}22)$$

Thus, Simplex is preempted if the value of the wake-up granularity parameter (*sysctl_sched_wakeup_granularity*) is less than the network sleep time *y* minus some small value ε, which corresponds to the amount of time the network process has spent running on the CPU. As we know that ε is greater than zero, eliminating the value of ε from formula (4-22) would not affect the inequality. Thus, Formula (4-22) can be simply written as:

$$sysctl\_sched\_wakeup\_granularity < y \quad (4\text{-}23)$$

Since the network process sleeps for a time duration less than 10 ms for more than 80% of the time, Simplex is more susceptible to be preempted by the network process when the value of the parameter *sysctl_sched_wakeup_granularity* is less than 10 ms. This explains why the performance of Simplex does not have a noticeable improvement when the value of the *sysctl_sched_wakeup_granularity* is increased from 1 ms to 10 ms.

- **Medium Wake-up Granularity Region.** In this range, the improvement in Simplex performance is very clear, as the Simplex execution time sharply drops from 109 s at 10

ms to 2.5 s at 20 ms. This trend is expected as more than 80% of the network process sleeps are shorter than 10 ms. Thus, when the value of the parameter *sysctl_sched_wakeup_granularity* exceeds 10 ms, Simplex becomes less susceptible to preemption by the waking up network process, as the condition in Formula (4-23) becomes harder to meet. Therefore, Simplex gets more CPU bandwidth, and its performance improves rapidly.

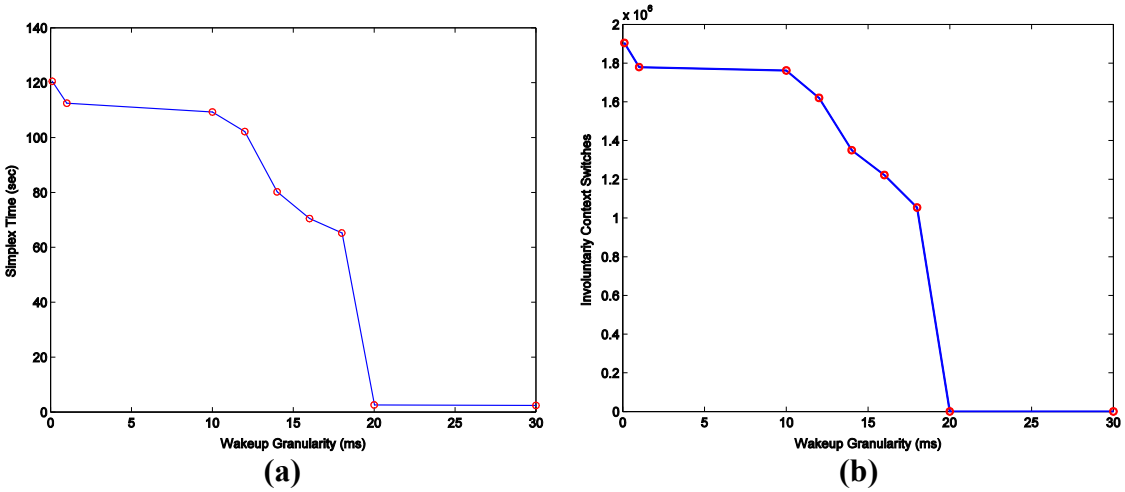

Figure 4.14: The effect of varying *sysctl_sched_wakeup_granularity* on (a) Simplex execution time and (b).Involuntary context switches

- **Large Wake-up Granularity Region.** Simplex performance does not have any noticeable improvement in this range, as its value reduces from 2.54 s at 20 ms to 2.36 s at 30 ms. This behavior is reasonable, as almost all the overhead caused to Simplex by the network process is eliminated when setting the value of the parameter *sysctl_sched_wakeup_granularity* to 20 ms. In fact, only 10 % of the network sleeps are longer than 15 ms. Thus, increasing the value of the parameter *sysctl_sched_wakeup_granularity* from 20 ms to 30 ms should not really have a major impact on the performance of Simplex.

The results in Figure 4.14(b), which show the number of involuntary context switches Simplex is forced to make while changing the value of the parameter *sysctl_sched_wakeup_granularity*, are very similar to the results in Figure 4.14(a). The improvement in Simplex performance is very small between 1 ms and 10 ms, as the reduction in the number of involuntary context switches of Simplex is very minimal. Then, the number of involuntary context switches sharply dropped between 10 ms and 20 ms, showing a great improvement in the performance of Simplex. Lastly, between 20 ms and 30 ms, the performance of Simplex is almost constant as the number of its involuntary context switches does not have a noticeable decrease in that range.

As the relation between the sleeping behavior of the network process and the performance degradation issue faced by CPU-bound processes becomes clear from the experiment results, we can understand the root cause of the performance bottleneck found in the problem assessment results (Figure 4.14). In fact, the problem assessment experiments were performed under the default setting of the Linux CFS, in which the default value of *sysctl_sched_wakeup_granularity* is 10 ms. As was earlier shown analytically (Formula (4-23)) and practically (Figure 4.14), 10 ms is not an appropriate value of *sysctl_sched_wakeup_granularity*, since it does not block the frequent preemption of Simplex by the network process.

The results of this experiment proved the conjecture about the relation between the performance of CPU-bound processes, represented by Simplex in the experiment, and the value of the parameter *sysctl_sched_wakeup_granularity*. As the value of the parameter increases, the performance of CPU-bound processes improves. Moreover, the results of the experiment have also proved the assumption that the sleeping behavior of

the network process is identical under both Linux 2.6 O(1) Scheduler and Linux CFS Scheduler. More importantly, the experiment results proved that the root cause of the performance degradation encountered by CPU-bound processes in networked Linux Systems running Linux CFS between 70 – 100 Kpps is actually the sleeping behavior of the network process, along with the inappropriate default setting of the scheduling parameter *sysctl_sched_wakeup_granularity* to 10 ms.

In this section, the performance problem CPU-bound processes face in networked Linux Systems running CFS was studied analytically and practically. The root cause of the problem was found to be the sleeping behavior of the network process, which tends to sleep too many times for very short time durations, in addition to the inappropriate default setting of the scheduling parameter *sysctl_sched_wakeup_granularity*. In Chapter 5, the emphasis will be on finding an appropriate solution to this performance degradation problem under Linux CFS.

CHAPTER 5

PROPOSED SOLUTION

5.1 Introduction

In Chapter 4, the performance issue faced by CPU-bound processes in a networked Linux system at a specific range of network load (i. e. 70 – 100 Kpps) was analyzed under two different versions of the Linux process scheduler, namely, Linux 2.6 O(1) Scheduler (kernel 2.6.16) and Linux 2.6 CFS (kernel 2.6.24). In this chapter, the emphasis is on finding appropriate solutions to the performance issue under both versions of the Linux Scheduler. Different solution approaches are proposed, studied and compared.

A systematic way for proposing solutions to the performance problem at hand is followed in this chapter. First, a simple global solution is started with, to give a good exposure to the solution domain. After studying this simple global solution, a generalized dynamic version of the solution is presented. The implementation and testing of the generalized local per-process solution is left for future work.

The rest of this chapter is organized as follows: section 5.2 presents solutions to the performance problem under Linux 2.6 O(1) Scheduler. Then, in section 5.2, solutions to the performance problem under Linux CFS are proposed and studied.

5.2 Solution under Linux 2.6 O(1) Scheduler

In Chapter 4, the analysis of the *sleep_time* value of the network process (*ITGRecv*) has shown that the sleeping behavior of the network process can cause the scheduler to boost its dynamic priority and treat it as an interactive process. This is because the scheduler design lacks the ability to recognize non-interactive processes that have many "short-term" sleeps. In addition, it was proved that this limitation in the scheduler design is the root cause for the observed instability in the performance of CPU-bound processes by filtering out the network process's "short-sleeps" and then studying how this affects the performance problem.

To come up with a general solution, we first start by a simple solution, in which we introduce to the scheduler a global threshold controlling the minimum sleep time (*sleep_time*) value that is allowed to be counted in the average sleep time of any process. The global solution is analyzed, with a special emphasis on its effects on the throughput of the network process. Finally, a generalized local (per-process) solution framework is proposed.

5.2.1 Global Solution

The lack of lower limits controlling the value of the sleep time (*sleep_time*) of any process in the Linux 2.6 O(1) scheduler design is a critical design issue that allows network processes with "short-sleeps" behavior to cause a severe instability in the performance of CPU-bound use applications. The simplest approach to solve this design

issue is to introduce a static global lower threshold on the value of the sleep time (*sleep_time*) of any process. In [CRA07], this static global lower threshold was implemented using the `/proc` filesystem. A similar implementation approach of this threshold is followed in this work.

The global solution of the observed performance problem in Linux 2.6 O(1) scheduler is implemented by modifying the Linux 2.6 scheduler code such that it includes an imposed globally-set static lower threshold on the value of the sleep time (*sleep_time*) of any process. In particular, the lower limit was named (*MIN_SLEEP_TIME_MS*) and was defined to be the lower limit for the *sleep_time* value in milliseconds. To be able to configure this lower limit from the user level, we defined an entry in the `/proc` filesystem through which the value of the *MIN_SLEEP_TIME_MS* can be modified instantly without a need to reboot the kernel. The entry was called *network_min_sleep*, which was defined under the `/proc/net` directory. To set the value of the *network_min_sleep* threshold, we use the `echo` shell command as follows:

```
#echo 10 > /proc/net/network_min_sleep
```

To read the current value of the *network_min_sleep* threshold, we use the `more` shell command as follows:

```
#more /proc/net/network_min_sleep
```

The complete solution can be found in Appendix C.

The value of the threshold was varied and analyzed in Chapter 4. The analysis results have shown that the best static value for the lower threshold is between 5 and 15 milliseconds. This agrees with the results presented in [CRA07], where a value of 5 ms

is recommended for machines connected through local area networks (LANs), as we have used a crossover cable connecting two machines in all of our experiments.

To ensure that the introduced lower threshold has no overhead and side-effects on the network process, we have to analyze the performance of the network process at different threshold values. For this purpose, we measured the network process performance using two quantitative values: the network throughput and latency. The network throughput in our experiments is measured as the number of packets that are received by the NIC and delivered to the application. The network latency is measured using the round trip time, i. e. the time it takes a packet to make a round trip from the sender to the recipient and then back to the sender.

The experimental setup used to measure the network performance was similar to Figure 3.1. The recipient side is loaded with the kernel in which the global solution is implemented. The sender sends a network stream to the recipient, and the recipient sends whatever it receives back to the sender. Thus, the measurements are recorded at the sender side. The recipient is running Simplex during the experiment, while the value of the *network_min_sleep* threshold is varied from 0 to 1000 ms. The two values, 0 and 1000, represent the two extreme ends of the process's *sleep_time* filtering process, namely, no filtering and exhaustive filtering.

The network process's throughput and delay results are shown in Figure 5.1 and Figure 5.2, respectively. Figure 5.1 shows the network throughput in Kpps against variable network load for different values of the *network_min_sleep* threshold. For the threshold values 0 ms, 15 ms, 30 ms, and 100 ms, there is almost no difference in the throughput of the network process. As the threshold value 0 represents the original

scheduler design, which does not have any lower limit on the value of *sleep_time*, having similar results for zero and nonzero threshold values proves that the introduced threshold has no negative impact on the throughput of the network process. Thus, the gain in the performance of Simplex when setting the threshold to 15 ms or 30 ms is not really causing degradation in the performance of the network process.

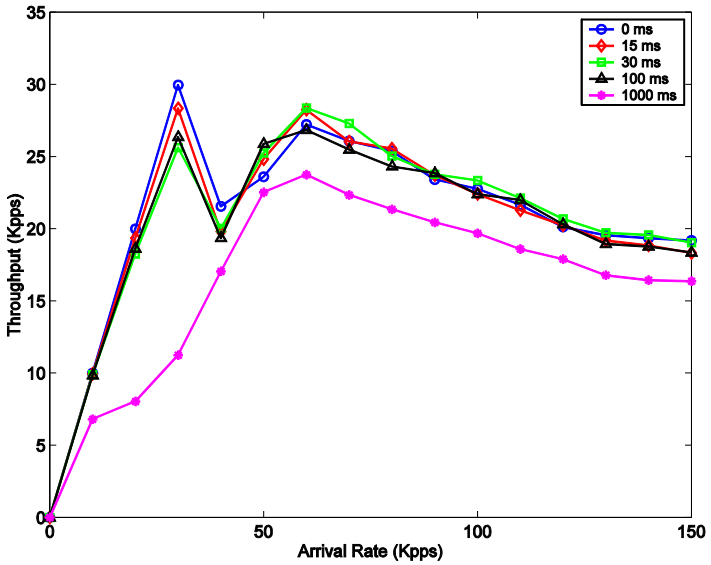

Figure 5.1: Network throughput at different threashold values

Similarly, the results in Figure 5.2 show no significant difference in the network process round-trip delay values when the threshold is set to 0 ms, 15 ms, 30 ms, or 100 ms. In particular, between 0 and 60 Kpps, the network round-trip time delay slightly increases when increasing the threshold value from 0 to 100 ms. However, beyond 60 Kpps, the difference in network round-trip time delay diminishes as the curves coincide over each other. Thus, our global solution, which recommends setting the threshold to 5 ms or 15 ms, has no significant impact on the network process round-trip delay. However, disabling the calculation of the process's sleep time (*sleep_time*) values, by setting the threshold value to 1000 ms, greatly affects the network process round-trip

delay. This is easily detectable from the results in Figure 5.2, as the curve corresponding to 1000 ms threshold value lies clearly above all other curves.

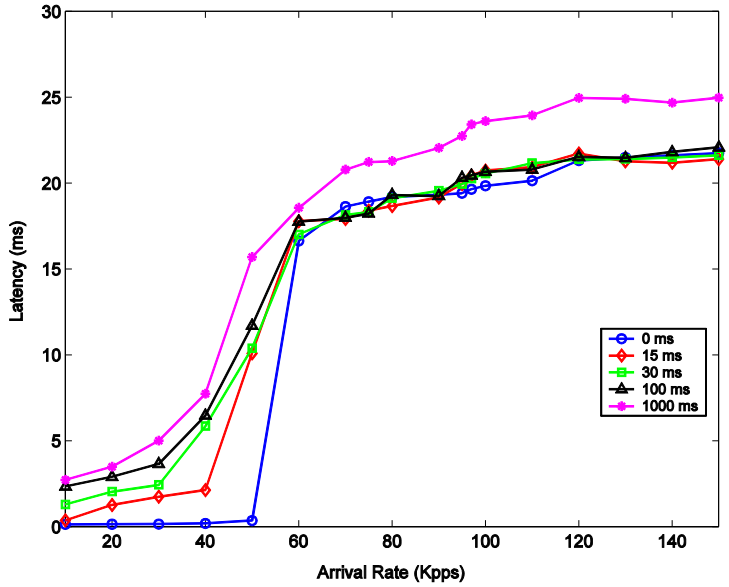

Figure 5.2: Network round trip delay at different threshold values

In conclusion, the performance analysis of the network process behavior under the introduced global solution shows that the solution has no negative impact on the performance of the network process. This was proved by studying the network process throughput and round-trip delay under different values of the network_min_sleep threshold. In addition, the performance analysis also shows that it is not recommended to disable the calculation of the sleep time (sleep_time) values of processes in a networked Linux system running Linux 2.6 O(1) Scheduler, as this would severely affect the performance of the networked applications.

5.2.2 Local Solution Framework

Solving the performance degradation issue faced by CPU-bound processes in a networked system running Linux 2.6 O(1) Scheduler using a global lower threshold limiting the (*sleep_time*) values of any process has many drawbacks. As processes have different natures and behaviors, treating them all with a single unique threshold would definitely affect the performance of some processes. In fact, in [CRA07], where the global solution was implemented and analyzed, it was clearly noted that the global solution might greatly affect certain network applications, such as multimedia network applications (e. g. VOIP applications), which usually sends and receives packets periodically. Thus, they recommend declaring such multimedia network processes as soft real time processes. However, such an approach, besides its complications, would greatly limit application portability [CRA07]. In addition, in [CRA07], it was shown that the network processes sleep time depends on the type of the network, being LAN or WAN, so the global threshold setting has to be updated accordingly. Therefore, the system end user has to be aware of all these complications to setup the global threshold to the appropriate value. Moreover, the observed performance problem is only present at a specific network traffic load, i. e. between 70 – 100 Kpps, a fact that the global solution approach simply ignores.

The drawbacks associated with the global solution are directly caused by the ignorance of the heterogeneity in the processes inherent characteristics, in addition to the ignorance of the different network conditions a networked system might encounter. In order to come up with a better solution approach, those important features have to be

given a special consideration and treated carefully. In particular, the solution approach has to be locally defined per process such that it accommodates all different network processes requirements under all network conditions, without any compromises. Those are the basic requirements for a better generalized solution approach.

In this section, we propose a local per-process solution framework that takes into account all of the above requirements. The solution is based on analyzing every non-interactive network process behavior at different network arrival rates, and then choosing the best minimum threshold that suits the process needs. Below is a general outline of the proposed local solution approach.

1. Declare a local *network_min_sleep* threshold for every non-interactive network process. This threshold would be most likely included in the task's structure of the non-interactive network process. It should have a default value of 0.
2. For every non-interactive network process, analyze the sleeping behavior of the process at different network arrival rates, starting from low rates up to high rates. In particular, collect the number of sleeps and the sleep durations of the network process while varying the network arrival rate.
3. Based on the collected statistics about the sleep behavior of the network process, identify the smallest value that most (around 90%) of the sleep duration samples fall below.
4. Set the default value of the *network_min_sleep* of the non-interactive network to the identified value in step 3.

5. If necessary, update the *network_min_sleep* threshold to the most appropriate value as special requirements arise, such as network traffic changes for certain non-interactive network processes that exhibits different sleeping behavior under different network traffic loads.

5.3 Solution under Linux CFS

The results of the experiment carried out in Chapter 4 to explore the relationship between the performance of CPU-bound processes, represented by Simplex, and the value of the parameter *sysctl_sched_wakeup_granularity* showed that it is an inverse relationship. As the value of the parameter increases, the performance of CPU-bound processes improves. Moreover, the results of the experiment have also proved the assumption that the sleeping behavior of the network process is identical under both Linux 2.6 O(1) Scheduler and Linux CFS Scheduler. Effectively, the experiment results proved that the root cause of the performance degradation encountered by CPU-bound processes in networked Linux Systems running Linux CFS between 70 – 100 Kpps is actually the sleeping behavior of the network process, along with the inappropriate default setting of the scheduling parameter *sysctl_sched_wakeup_granularity* to 10 ms.

In this section, the emphasis is on finding an appropriate solution to the performance degradation problem under Linux CFS. We will start by presenting a simple global solution to the problem, and then we will improve the solution to be a more generalized local (per-process) solution.

5.3.1 Global Solution

Linux CFS comes with several tuning parameters which can be set through the *sysctl* command. Examples of those parameters include the *sysctl_sched_min_granularity* parameter, which is used to define the minimal preemption granularity for CPU-bound tasks, *sysctl_sched_latency* parameter used to define scheduling period in CFS algorithm, *sysctl_sched_child_runs_first*, which is used to decide which one of the parent process and its forked child should run first. All of those parameters come with default settings that are usually the most recommended settings for a general use of Linux.

sysctl_sched_wakeup_granularity is one of those Linux CFS parameters which has a special importance under the *SCHED_OTHER* scheduling policy, as it is used to control the preemption effect of decoupled normally-scheduled workloads, and reduce their over-scheduling. The default setting of this parameter is 10 ms in kernel 2.6.24. As the CFS is fairly new and being continuously modified and improved, the default setting of *sysctl_sched_wakeup_granularity* has been changed in kernel 2.6.27 to 5 ms [LKA].

The performance problem theoretical and experimental analysis, presented in Chapter 4m has shown that the most recommended setting of the parameter *sysctl_sched_wakeup_granularity* that would solve the performance problem at hand should follow the rule in Formula (4-23):

$$sysctl\_sched\_wakeup\_granularity < y \,, \qquad (4\text{-}23)$$

where y is the network process sleep time value, which is mostly between 5 ms and 15 ms when the network traffic rate is in the critical range of 70 – 100 Kpps. Thus, neither

the old nor the new default setting of *sysctl_sched_wakeup_granularity* satisfies the derived rule in Formula (4-22).

A simple solution to the performance problem under Linux CFS is to modify the default setting of *sysctl_sched_wakeup_granularity* to be greater than most of the network process sleep time values when the network traffic rate is between 70 – 100 Kpps. This means that we have to modify the default setting the value of *sysctl_sched_wakeup_granularity* to be greater than 15 ms. From the analysis results in Chapter 4, it was shown that there is no noticeable gain in performance when the value of *sysctl_sched_wakeup_granularity* exceeds 20 ms. Thus, in this solution, we modify the default setting of the parameter *sysctl_sched_wakeup_granularity* to be 20 ms. This is easily implemented by changing the value in the scheduling module *sched_fair.c*, see Figure 5.3.

```
72 /*
 73  * SCHED_OTHER wake-up granularity.
 74  * (default: 10 msec * (1 + ilog(ncpus)), units: nanoseconds)
 75  *
 76  * This option delays the preemption effects of decoupled workloads
 77  * and reduces their over-scheduling. Synchronous workloads will still
 78  * have immediate wakeup/sleep latencies.
 79  */
 80 /*unsigned int sysctl_sched_wakeup_granularity = 10000000UL;*/

 81 unsigned int sysctl_sched_wakeup_granularity = 20000000UL;
```

Figure 5.3: Extract from sched_fair.c in kernel 2.6.24 which shows the place at which the default value of sysctl_sched_wakeup_granularity can be modified

It is important to study how this new default setting would affect the performance of the network process. Similar to our work in section 5.1.1, we measured the network process performance using two quantitative values: the network throughput and latency. The network throughput in our experiments is measured as the number of packets that are received by the NIC and delivered to the application. The network latency is measured using the round trip time, i. e. the time it takes a packet to make a round trip from the sender to the recipient and then back to the sender.

To analyze and compare the performance of the network process at different values of the parameter *sysctl_sched_wakeup_granularity*, we setup an experiment similar to the one used in section 5.2.1. The only difference between the experiment in 5.1.1 and this experiment is that this time the recipient is running kernel 2.6.24, and that at the recipient we are varying the value of *sysctl_sched_wakeup_granularity* instead of `network_min_sleep`. The results of the experiment are shown in Figure 5.4 and 5.5.

Figure 5.4 shows the performance of the network process in terms of throughput as we vary the value of the parameter *sysctl_sched_wakeup_granularity* from 1 ms to 30 ms, passing by the default value in kernel 2.6.24, i. e. 10 ms. As all the graphs of the network throughput at different values of *sysctl_sched_wakeup_granularity* coincide over each other, we conclude that there is actually no effect on the throughput of the network process when changing the default value of the parameter from 10 ms to 20 ms.

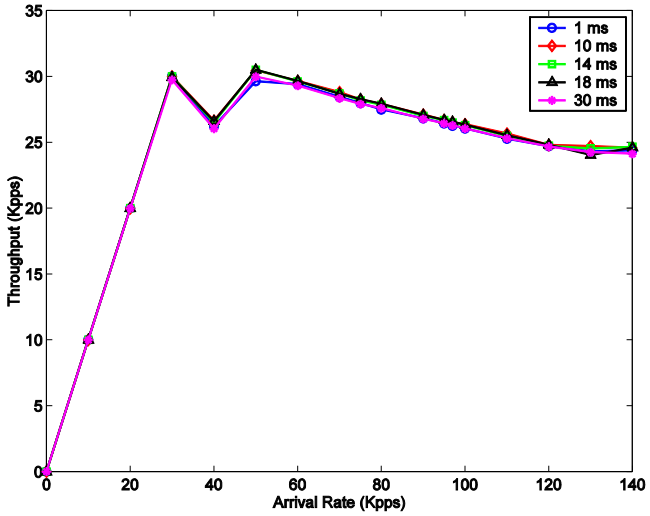

Figure 5.4: Network throughput at different values of *sysctl_sched_wakeup_granularity*

On the other hand, Figure 5.5 shows the behavior of the network performance in terms of round-trip time delay when varying the value of the parameter *sysctl_sched_wakeup_granularity*. Between 0 and 40 Kpps, the network round-trip time delay sharply increases, while beyond 40 Kpps the increase in delay becomes much less. Although there are some differences in the delay when changing the value from 1 ms to 30 ms, the differences are very small. Indeed, beyond 30 Kpps, the graphs almost coincide over each other, showing nearly no difference in the network round trip time delay. Thus, the overall results in Figure 5.5 show no significant change in the network round-trip time delay when changing the default value of the parameter from 10 ms to 20 ms.

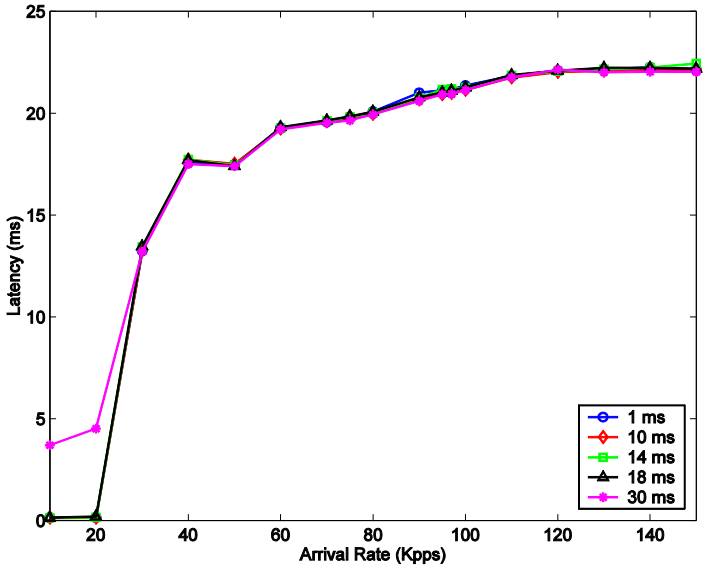

Figure 5.5: Network round trip delay at different values of *sysctl_sched_wakeup_granularity*

In conclusion, the performance analysis of the network process behavior under the introduced global solution shows that the solution has no negative impact on the performance of the network process. This was proved by studying the network process throughput and round-trip delay under different values of the *sysctl_sched_wakeup_granularity*. Thus, the above results highly recommends changing the default setting of the CFS parameter *sysctl_sched_wakeup_granularity* from 5 ms or 10 ms to some value greater than 15 ms, say 20 ms.

5.3.2 Local Solution Framework

The global solution presented in the previous section is not an ultimate solution to the performance degradation issue faced by CPU-bound processes in a networked system running Linux CFS, as it has many limitations. In fact, processes, even those scheduled under the same scheduling class, are heterogeneous in their natures and

behaviors. Therefore, having a global unique value for the parameter *sysctl_sched_wakeup_granularity* might not suit the needs of all processes. As mentioned earlier in section 5.2.2, some network applications, such as multimedia network applications, usually sends and receives packets periodically. Thus, they have a special sleeping behavior, which might be affected by increasing or decreasing the value of *sysctl_sched_wakeup_granularity***.** In addition, the differences in the network environment, being LAN or WAN, as well as the differences in the network load at different times, are very significant factors in shaping the sleeping behavior of many network process [CRA07]. Those factors are simply ignored when setting a unique global value for the parameter *sysctl_sched_wakeup_granularity*.

To come up with a better approach for handling the value of the parameter *sysctl_sched_wakeup_granularity*, we need to take into account the heterogeneity in the processes inherent characteristics, in addition to the different network conditions a networked system might encounter. Moreover, the system end-uses might not be aware of all these complications, so the solution approach has to be dynamic and changes the value of *sysctl_sched_wakeup_granularity* automatically when the network conditions changes, without any need for the user intervention. Those are the basic requirements for a better generalized solution approach.

To satisfy all of those requirements, a local solution approach is proposed in this section. The basic idea of the solution is to make the parameter *sysctl_sched_wakeup_granularity* specific per process, and base its value on the process's sleeping behavior. As the basic concept of *sysctl_sched_wakeup_granularity* is to delay the preemption effects of decoupled workloads and avoid over-scheduling them,

it is logical to relate the value of the parameter to the sleeping behavior of the process. Thus, we defined a specific wakeup granularity parameter per process, whose value depends on the number of sleeps per scheduling period, in addition to the process mapped priority (nice) value, i. e. the scheduling entity load weight. We propose the following formula for calculating the per-process wakeup granularity parameter:

$$se \rightarrow sched\_wakeup\_granularity = \frac{\alpha \times se \rightarrow num\_sleeps}{sched\_period \times se \rightarrow load.weight}, \qquad (5\text{-}1)$$

where

$se \rightarrow sched\_wakeup\_granularity$ is the per-process wakeup granularity parameter,

$se \rightarrow num\_sleeps$ is the number of sleeps of the process per scheduling period,

$sched\_period$ is the scheduling period,

$se \rightarrow load.weight$ is the mapped nice value of the process, and

α is a positive constant used as a scaling factor.

The solution approach is very simple. For each process, we track the number of sleeps it makes for every scheduling period. As the number of sleeps per scheduling period increases, the *sched_wakeup_granularity* value increases, with a constant scaling factor. This ensures that processes which tends to sleep too many times in very short time, i. e. the scheduling period, are being penalized gradually for their sleeping behavior. In addition, the process's priority is taken into account, to give a special consideration for high-priority processes. At the beginning of the next scheduling period, the number of sleeps of each process is set back to zero. The detailed solution algorithm is given below.

Algorithm: Per-Process sched_wakeup_granularity

Input: se→ num_sleeps: the number of sleeps the scheduling entity makes during sched_period time
sched_period: the scheduling period time, during which CFS tries to run each task once
se→load.weight: the weight of the scheduling entity se, mapped from its nice value.
α: a positive constant

Output: se→sched_wakeup_granularity: the wakeup granularity parameter of the scheduling entity se:

se→num_sleeps ← 0
α ← some positive constant

1. When se wakes up from a sleep
   se→ num_sleeps = se→ num_sleeps + 1

   calculate the new value of se→sched_w\akeup_granularity using the formula:

   $$se \rightarrow sched\_wakeup\_granularit\ y = \frac{\alpha \times se \rightarrow num\_sleeps}{sched\_period \times se \rightarrow load.weight}$$

2. For every new sched_period
   se→num_sleeps ← 0

In this chapter, we have presented solutions to the observed performance problem under both version of the Linux Scheduler, namely, Linux 2.6 O(1) Scheduler, and Linux CFS. In both versions, we first presented a simple global solution approach and analyzed its performance. After that, we devised a more generalized local solution approach that eliminates the shortcomings of the first simple global approach. The implementation and testing of the local solution approaches are left for future work.

CHAPTER 6

PERFORMANCE OF CPU-BOUND PROCESSES UNDER SMP CONFIGURATION

6.1 Introduction

Recently, the development of CPU's that have multiple processing units, so-called cores, have noticed a great advancement, such that almost all simple desktop machines have CPU's that run multiple cores. This revolution in hardware multiprocessing technologies has motivated operating systems developers to redesign their operating systems such that they make use of the available hardware multiprocessing power. Thus, the importance of and demand for operating systems that are built using symmetric multiprocessing (SMP) concepts is constantly increasing.

In chapters 3, 4 and 5, we studied how frequent interrupts caused by received or transmitted packets in a networked Linux system may stretch the execution and eventually starve user processes. We assessed the problem scope in Chapter 3, analyzed its behavior in Chapter 4, and suggested solutions to the problem in Chapter 5. However, our main focus in all of these chapters was only on systems running on a single processor, i. e. uni-processor Linux systems. In this chapter, we give a brief look at the problem in multiprocessing computing environments.

The remainder of this chapter is organized as follows: Section 6.2 illustrates the experimental setup used for studying the problem under Linux SMP environment. Then, in Section 6.3, we look at the performance problem in SMP-configured Linux 2.6 O(1) scheduler of kernel 2.6.16. Finally, we look at the problem in the Linux SMP CFS of kernel 2.6.24 in Section 6.4.

6.2 Experimental Setup

To study the performance problem under SMP environment, we used an experimental setup similar to that in Figure 3.1. The sender runs Linux kernel 2.6.16, while the recipient runs Linux kernel 2.6.16 in the first experiment (Section 6.2) and Linux kernel 2.6.24 in the second (Section 6.3). In both experiments, the kernel is configured with SMP options, as the recipient runs on Pentium 3.2GHz Core 2 Due processor. We used open source *Distributed Internet Traffic Generator* (D-ITG) to generate network traffic at a rate of 75 Kpps in both experiment.

As the recipient has two processing cores, the problem would not be visible in anyway by running only one Simplex process while receiving the traffic. This is because the scheduler usually runs the network process on one core and the Simplex process on the other. This can be easily detected by using the `mpstat` command with the option `-P ALL`, as follows:

```
$ mpstat -P ALL
```

Thus, to make the CPU-bound user process and the network process compete on the same core, we have to run more than one Simplex process. In this experiment, we run

two Simplex processes: one Simplex process occupies one core for the whole experiment, and the other Simplex process competes with the network process (i. e. *ITGRecv*) on the other core. We are interested in the performance of the second Simplex process.

6.3 Linux 2.6 O(1) Scheduler

To study whether the performance problem is present in Linux 2.6 O(1) Scheduler configured with SMP option, we used the experimental setup described in 6.2 to generate a network traffic at a rate of 75 Kpps from the sender to the recipient. The recipient is running two Simplex processes, one is hogging one of the processor cores, and the other is competing with *ITGRecv* network process on the other core. We verified the distribution of processes on cores using the `mpstat` command. The recipient is running Linux kernel 2.6.16 which includes an implementation of the global solution described in Chapter 5 Section 5.2.1. Thus, while we are sending the traffic, we varied the value of the *network_min_sleep* threshold, from 0 ms to 1000 ms. The two values represent the two extreme ends of the process's *sleep_time* filtering process, namely, no filtering and exhaustive filtering. The experiment results are shown in Figure 6.1.

As apparent from the result in Figure 6.1(a), the performance problem is clearly present under Linux 2.6 O(1) Scheduler configured with SMP option, as the time Simplex takes to run calculations needing only 1 second on a free CPU take almost 40 seconds when there is no filtering of the *sleep_time* values of the network process (i. e. *network_min_sleep* threshold is set at 0). When the *network_min_sleep* threshold is

increased from 0 to 20 ms, we notice a sharp decrease in the time Simplex takes, reducing from 38 seconds to 7 seconds. Beyond the threshold value of 20 ms, the graph gradually gets flattened, as there is no significant decrease in Simplex execution time. Similarly, the results in Figure 6.1(b), (c), (d) follow the same trend. Moreover, the result in Figure 6.1(e) shows how the amount of CPU time available for Simplex increases sharply between 0 ms and 20 ms threshold values, then the amount of gain in CPU time gradually decreases beyond 20 ms, giving the graph its logarithmic shape.

Several important conclusions can be drawn from the above experiment results. First, the performance problem is present under Linux SMP environment running Linux 2.6 O(1) Scheduler. In addition, the sleeping behavior of the network process under SMP is very similar to its sleeping behavior in uni-processor environments. This can be easily concluded from the sharp drop in the graph between threshold values of 0 ms and 20 ms, which indicates that most of the network process sleeps under SMP are of length shorter than 20 ms. Finally, the global solution introduced in Chapter 5 Section 5.2.1 for uni-processor Linux environments is also effective in solving the performance problem under Linux SMP environments.

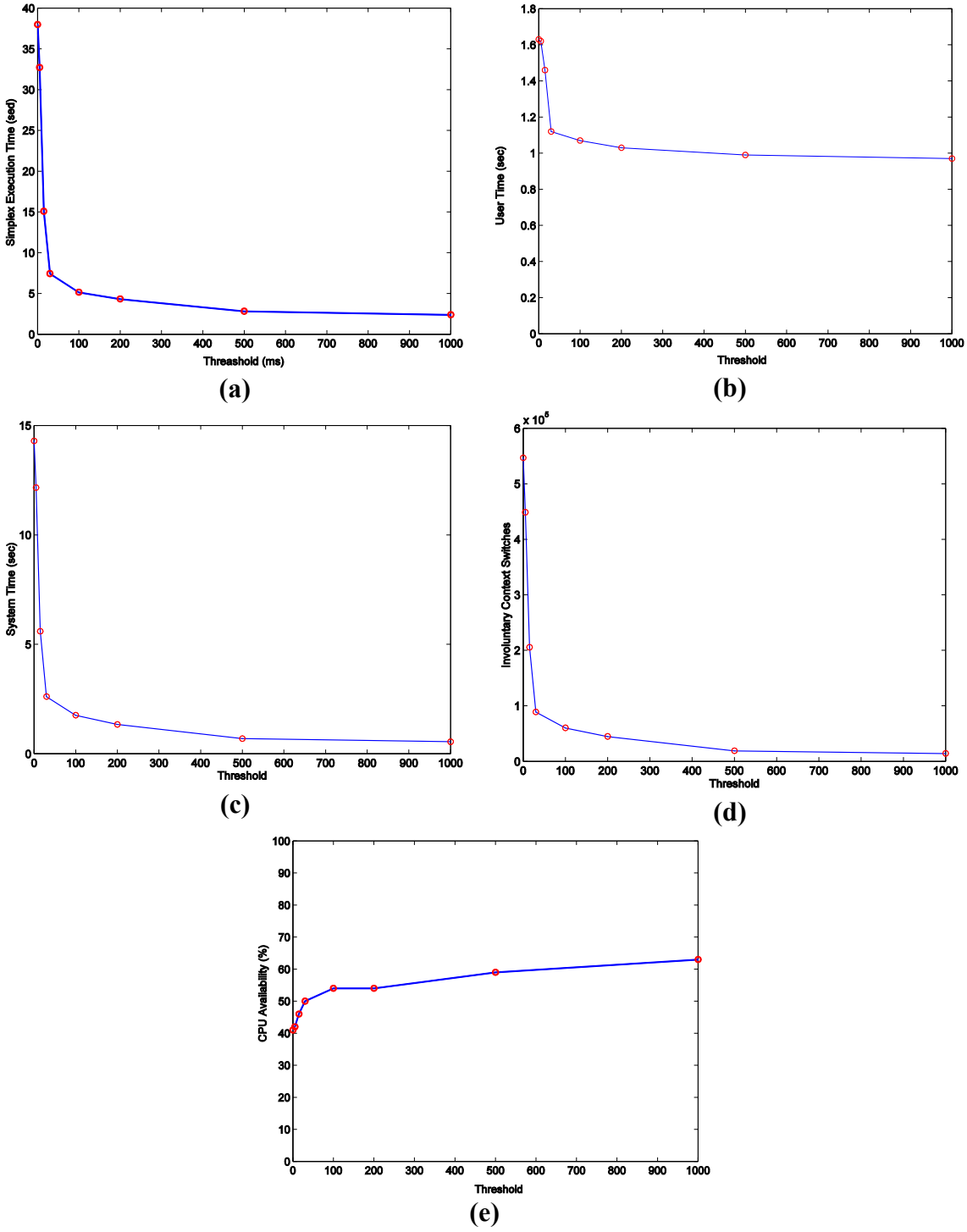

Figure 6.1: Performance of Simplex under kernel 2.6.16 configured with SMP option, with different values of the *NETWORK_MIN_SLEEP* threshold

6.4 Linux CFS

The presence of performance problem under Linux CFS configured with SMP option is assessed using the experimental setup described in 6.2. We generated network traffic at a rate of 75 Kpps from the sender to the recipient. At the recipient side, two Simplex processes are running. One Simplex process is running on one of the processor cores, and the other is competing with *ITGRecv* network process on the other core. The distribution of the processes among the two cores was verified using the `mpstat` command. The recipient is running Linux kernel 2.6.24. While we are sending the traffic, we varied the value of the scheduling parameter *sysctl_sched_wakeup_granularity* from 1 ms to 100 ms using the command:

```
# sysctl -w kernel.sysctl_sched_wakeup_granularity="10000000"
```

The performance of the second Simplex process is recorded for every value of the parameter. The results are shown in Figure 6.2.

Figure 6.2(a) shows that the performance problem is present under Linux CFS configured with SMP option. However, the severity of the problem is much lower than its severity under uni-processor environments, as the time Simplex takes to run calculations needing only 1 second on a free CPU is 3.7 seconds when the parameter *sysctl_sched_wakeup_granularity* was set to its default value of 10 ms. Moreover, comparing this experiment results with the results of the similar experiment carried out to investigate the performance problem in uni-processor CFS (Figure 4.12), it is clear that the results of the two experiments agree on having the most significant reduction of Simplex execution time between parameter values of 1 ms and 20 ms. Beyond parameter

value of 20 ms, the decrease in Simplex execution time becomes less apparent, as the graph gets flattened.

Similarly, Figures 6.2(c) and (d), which show the system time portion of Simplex execution time and the involuntary context switches of Simplex, respectively, follow the same trend in Figure 6.2(a). However, the user time shown in Figure 6.2(b) has a different behavior, where the user time is almost constant around 1. second, with small oscillations of no more than 0.1 second difference. Moreover, the CPU resources available to Simplex, shown in Figure 6.2(e), has a logarithmic shape that supports the fact that the most significant gain in Simplex performance is observed when the value of the parameter *sysctl_sched_wakeup_granularity* is increased from 1 ms to 20 ms. In addition, setting the *sysctl_sched_wakeup_granularity* parameter to a value greater than 30 ms most would most likely affect the network process performance, as Simplex is observed to unjustifiably gain more than 90% of the CPU resources.

The above experiment results highlight several important findings. First, the performance problem is present under Linux CFS configured with SMP option, though its severity is much less. In addition, the sleeping behavior of the network process under SMP environment is similar to its sleeping behavior under uni-processor environment. Finally, the new recommended default setting of the parameter *sysctl_sched_wakeup_granularity* to 20 ms, presented in Chapter 5 as a global solution to the performance problem under CFS configured with uni-processor option, is also effective in solving the problem under SMP-configure CFS.

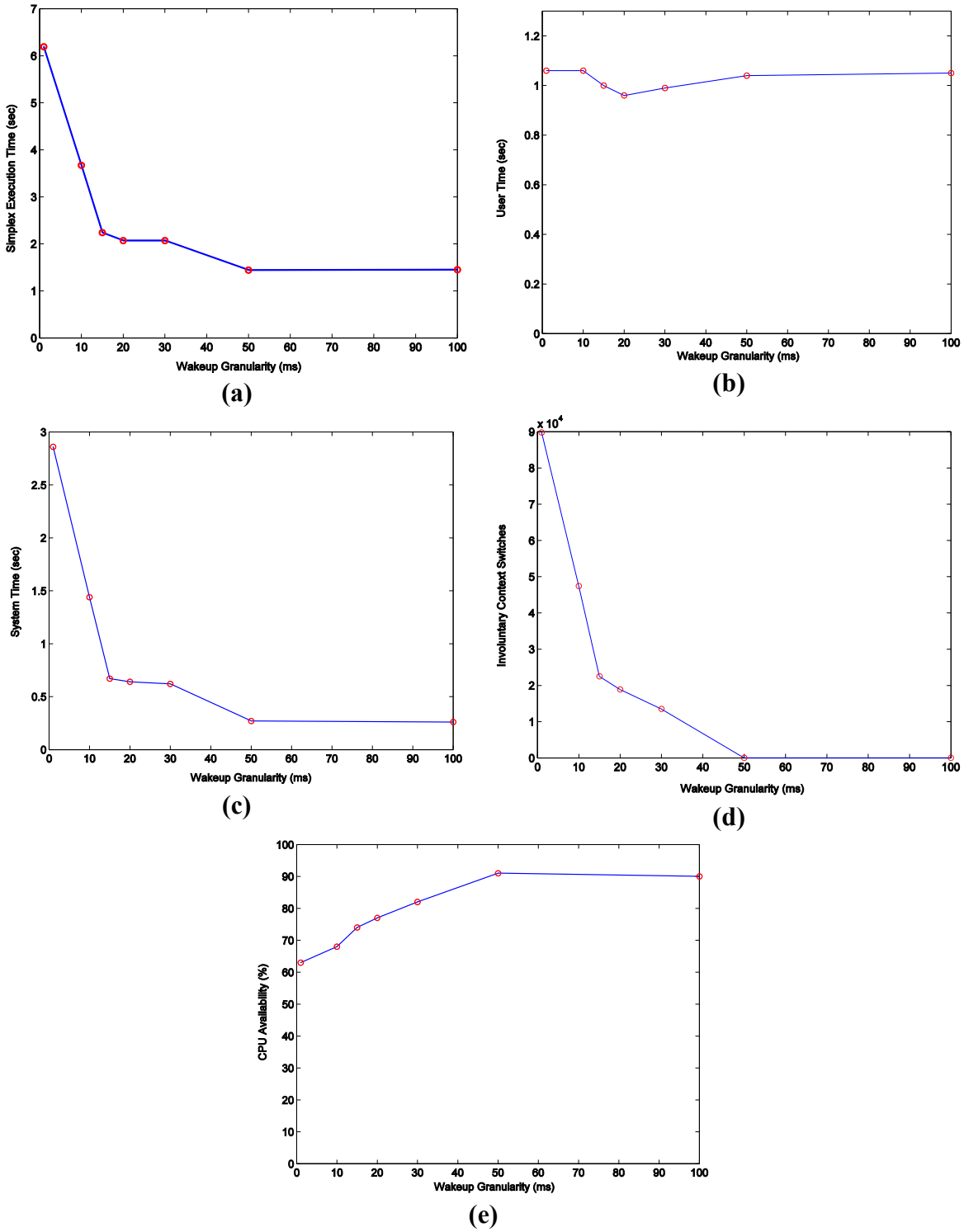

Figure 6.2: Performance of Simplex under kernel 2.6.24 configured with SMP option, with different values of the *sysctl_sched_wakeup_granularity* scheduling parameter.

In this chapter, we have given a brief look at the performance problem under Linux SMP environment of both the old 2.6 O(1) Scheduler and the new Completely Fair Scheduler (CFS). The performance problem was found in both versions of the scheduler, with varying severity. The network sleeping behavior in SMP environment was found to be almost identical to its behavior in uni-processor environment. The global solutions introduced in Chapter 5 were found to be effective in solving the performance problem under Linux SMP environments.

CHAPTER 7

CONCLUSION

In this chapter, we present a summary of the major contributions in this thesis work to study the bottleneck in the performance of CPU-bound user processes in a networked Linux system. It also gives indications of future research directions.

One of important contributions of this research work is the assessment of the scope of the performance degradation issue faced by CPU-bound user processes in a networked Linux environment. The performance was assessed under different kinds of network I/O-bound processes, different types of network interface cards, and different versions of the Linux kernel.

Another major contribution of this research is the identification of the root cause of the performance instability problem encountered by CPU-bound processes in a networked Linux system running 2.6 O(1) scheduler. The root cause of the problem was found to be the inability of the scheduler to correctly classify the network process as a non-interactive process. This is because the scheduler design lacks the ability to recognize the "frequent-short-sleeping" behavior of the network process, which allows it to get full advantage of all of its non-interactive sleeps, and thus gets a boosted priority.

Similarly, the performance problem CPU-bound processes face in networked Linux Systems running CFS was studied analytically and practically. The root cause of

the problem was found to be the sleeping behavior of the network process, which tends to sleep too many times for very short time durations, in addition to the inappropriate default setting of the scheduling parameter *sysctl_sched_wakeup_granularity*.

Moreover, we have presented solutions to the observed performance problem under both version of the Linux Scheduler, namely, Linux 2.6 O(1) Scheduler, and Linux CFS. In both versions, we first presented a simple static approach and analyzed its performance. After that, we devised a better local per-process solution approach that eliminates the shortcomings of the first simple approach.

Finally, we have given a brief look at the performance problem under Linux SMP environment of both the old 2.6 O(1) Scheduler and the new Completely Fair Scheduler (CFS). The performance problem was found in both versions of the scheduler, with varying severity. The network sleeping behavior in SMP environment was found to be almost identical to its behavior in uni-processor environment. The global solutions introduced in Chapter 5 were found to be effective in solving the performance problem under Linux SMP environment.

The work presented in this thesis opens many horizons for future research. The following are some future directions:

- The effectiveness and performance of the local per-process solution approaches introduced in this research should be assessed using simulation. The simulation study would give an insight into how useful both local per-process solution approaches are in solving the observed performance problem under Linux 2.6 O(1) Scheduler and Linux CFS. In addition, simulating both

solution approaches using different values for their parameters would also help in identifying the most appropriate parameter settings that should be used for both solutions. Moreover, the simulation study could also identify any side effects associated with both solution approaches.

- Based on the simulation study results, the proposed solution frameworks could be implemented and experimentally assessed. The effectiveness and performance of both solutions should be studied and analyzed under different system configuration and using different kinds of NIC's. The performance of both solutions should be also analyzed using different kernel benchmarks. Based on the results of this experimental study, the solutions might be proposed to the Linux development community in a form of a patch.
- In this research, we have only given a brief look at the performance problem under Linux SMP environment. A more detailed analysis of the problem under Linux SMP environment is needed. As SMP environments are very different from uni-processor environments, the proposed local per-process solution approaches might need some modification to work properly in SMP systems. Both simulation and practical implementation should be used in the analysis of the solution approaches under SMP environment.
- In this research, we have only focused on one scheduling policy, namely, the normal desktop scheduling policy, i. e. *SCHED_OTHER* in 2.6 O(1) Scheduler and *sched_fair* in CFS. This research work might be extended by studying the performance of CPU-bound user processes in a networked

Linux system under different kinds of scheduling policies. Moreover, the performance of CPU-bound user processes could be also studied under the group scheduling module that came along with Linux CFS.

APPENDIX A

INSTRUMETATION OF LINUX 2.6 O(1) SCHEDULER TO DISABLE DYNAMIC-PRIORITY CALCULATION

To disable the dynamic priority calculation in the Linux 2.6 O(1) scheduler, we modified the *recalc_task_prio()* function inside the kernel scheduler module (*sched.c*) such that it does not do any updates to the average sleep value of any process, i. e. *sleep_avg*. As the dynamic priority calculation of any process is directly dependent on the value of the process's average sleep time (*sleep_avg*) [1], disabling the calculation of the process's average sleep time has disables, in effect, the dynamic priority calculation. Thus, we disabled the dynamic priority calculation in the scheduler thru disabling all the calculation of, or more precisely, the updates to the average sleep time of any process. We built the 2.6.16 kernel with this modified scheduler. Below is the code of the modified *recalc_task_prio()* function. The modified code portions are shown in ***bold italics***.

[1] Look at section 4.1.2 for more details.

```
static int recalc_task_prio(task_t *p, unsigned long long now)
{
      /* Caller must always ensure 'now >= p->timestamp' */
      unsigned long long __sleep_time = now - p->timestamp;
      unsigned long sleep_time;

      if (unlikely(p->policy == SCHED_BATCH))
            sleep_time = 0;
      else {
            if (__sleep_time > NS_MAX_SLEEP_AVG)
                  sleep_time = NS_MAX_SLEEP_AVG;
            else
                  sleep_time = (unsigned long)__sleep_time;
      }

      if (likely(sleep_time > 0)) {
            /*
             * User tasks that sleep a long time are categorised as
             * idle and will get just interactive status to stay active
&
             * prevent them suddenly becoming cpu hogs and starving
             * other processes.
             */
            if (p->mm && p->activated != -1 &&
                  sleep_time > INTERACTIVE_SLEEP(p)) {
                        p->sleep_avg = JIFFIES_TO_NS(MAX_SLEEP_AVG -
                                    DEF_TIMESLICE);
            } else {
                  /*
                   * The lower the sleep avg a task has the more
                   * rapidly it will rise with sleep time.
                   */
                  sleep_time *= (MAX_BONUS - CURRENT_BONUS(p)) ? : 1;

                  /*
                   * Tasks waking from uninterruptible sleep are
                   * limited in their sleep_avg rise as they
                   * are likely to be waiting on I/O
                   */
                  if (p->activated == -1 && p->mm) {
                        if (p->sleep_avg >= INTERACTIVE_SLEEP(p))
                              sleep_time = 0;
                        else if (p->sleep_avg + sleep_time >=
                                    INTERACTIVE_SLEEP(p)) {
                              p->sleep_avg = INTERACTIVE_SLEEP(p);
                              sleep_time = 0;
                        }
                  }

                  /*
                   * This code gives a bonus to interactive tasks.
                   *
                   * The boost works by updating the 'average sleep
time'
```

```
                        * value here, based on ->timestamp. The more time a
                        * task spends sleeping, the higher the average gets
-
                        * and the higher the priority boost gets as well.
                        */
                ***/* Disable Updates to p->sleep_avg*/***
                      ***/*p->sleep_avg += sleep_time; */***

                      if (p->sleep_avg > NS_MAX_SLEEP_AVG)
                            p->sleep_avg = NS_MAX_SLEEP_AVG;
                }
        }

        return effective_prio(p);
        }
```

APPENDIX B

PROFILING THE SLEEP TIME (*sleep_time*) VALUES OF THE NETWOR PROCESS (*ITGRecv*)

For profiling the sleep time (*sleep_time*) values of the network process (*ITGRecv*), we introduced an instrumentation code in the Linux kernel such that we can trace the sleep time of the process *ITGRecv* during different values of network load. As it is almost impossible to trace every single value of the sleep time of the network process, since this will leave the system almost unusable due to the extra overload on the system scheduler, we designed the instrumentation code to use just few representative samples of the sleep time of the network process. The sample collection process is triggered and terminated externally from the user space. The instrumentation code was introduced to the scheduler code without affecting its original control flow.

The basic design of the collection process is to temporarily store every single value of sleep time (*sleep_time*) the network process (*ITGRecv*). The storage, which is basically a long variable called *ITGRecv_sleep_sample*, is defined globally. Whenever the network process (*ITGRecv*) wakes up from a sleep, the new value of the *sleep_time* is stored in *ITGRecv_sleep_sample*. We also designed an interface to the user space using the `/proc` filesystem. We defined a new entry in the proc filesystem under `/proc/net`, which we call *ITGRecv_sleep_sample*. We placed the code to create this proc entry inside the *enqueue_task()* function, such that the entry is created when the

first task in the system is inserted into the runqueue. To read the current sleep sample, we just query this entry using the `more` shell command as follows:

```
#more /proc/net/ITGRecv_sleep_sample
```

Thus, to automate the sample collection process, we created a small shell script to collect the samples and store them in a log file. We used this shell script to collect more than 500 samples. It is important to notice that though the scheduler is very fast, such that it is very hard to read the same sample value twice, we ensured that this would not happen by zeroing out the sample value after it has been queried from the user space. Thus, we make sure that we do not end up reading the same sample value more than one time. We did this modification under the kernel 2.6.16. The code and script used to implement this sample collection logic is shown below. The modified code portions are shown in ***bold italics***.

1. Modified Scheduler Code:

```
/* Additional Includes */
#include <linux/proc_fs.h>
#include <linux/string.h>
#include <linux/kernel.h>
#include <asm/unistd.h>

/* Global Variables */
static struct proc_dir_entry *proc_entry;
static int startup = 0;
static unsigned long itg_avg=0;

/*Additional Function Reading The Proc Entry */

int itg_avg_read( char *page, char **start, off_t off,

                  int count, int *eof, void *data )

{
   int len = sprintf(page, "\n*********\nITGRecv Sleep Sample = %lu\n
", itg_avg);
   printk(KERN_INFO "\n*********\nITGRecv Sleep Sample = %lu\n ",
itg_avg);
   itg_avg=0;
   return len;
}

/* Modified enqueu_task Function, Which Contains the Code to Create the
Proc Filesystem Entry*/
static void enqueue_task(struct task_struct *p, prio_array_t *array)
{
      sched_info_queued(p);
      list_add_tail(&p->run_list, array->queue + p->prio);
      __set_bit(p->prio, array->bitmap);
      array->nr_active++;
      p->array = array;

      /*Additional Block*/
      if(startup == 0)
      {
            startup = 1;
            int ret = 0;
            option_buffer = (char *)vmalloc(1024);
            memset(option_buffer, 0, 1024);
            proc_entry = create_proc_entry("ITGRecv_sleep_sample",
0444, proc_net );
            if (proc_entry == NULL) {
                        ret = -ENOMEM;
                        printk(KERN_INFO "***** Couldn't create proc
entry ITGRecv_sleep_sample \n");
            } else {
                        proc_entry->read_proc = itg_avg_read;
```

```
                        printk(KERN_INFO "***** proc entry
ITGRecv_sleep_sample successfully created\n");

            }
      }
}


/* Modified recalc_task_prio Function, Which Contains the Code to store
the sleep_time samples of ITGRecv*/
static int recalc_task_prio(task_t *p, unsigned long long now)
{
      /* Caller must always ensure 'now >= p->timestamp' */
      unsigned long long __sleep_time = now - p->timestamp;
      unsigned long sleep_time;

      if (unlikely(p->policy == SCHED_BATCH))
            sleep_time = 0;
      else {
            if (__sleep_time > NS_MAX_SLEEP_AVG)
                  sleep_time = NS_MAX_SLEEP_AVG;
            else
                  sleep_time = (unsigned long)__sleep_time;
      }

      if (likely(sleep_time > 0)) {
            /*
             * User tasks that sleep a long time are categorised as
             * idle and will get just interactive status to stay active
&
             * prevent them suddenly becoming cpu hogs and starving
             * other processes.
             */
            if (p->mm && p->activated != -1 &&
                  sleep_time > INTERACTIVE_SLEEP(p)) {
                        p->sleep_avg = JIFFIES_TO_NS(MAX_SLEEP_AVG -
                                    DEF_TIMESLICE);
            } else {
                  /*
                   * The lower the sleep avg a task has the more
                   * rapidly it will rise with sleep time.
                   */
                  sleep_time *= (MAX_BONUS - CURRENT_BONUS(p)) ? : 1;

                  /*
                   * Tasks waking from uninterruptible sleep are
                   * limited in their sleep_avg rise as they
                   * are likely to be waiting on I/O
                   */
                  if (p->activated == -1 && p->mm) {
                        if (p->sleep_avg >= INTERACTIVE_SLEEP(p))
                              sleep_time = 0;
                        else if (p->sleep_avg + sleep_time >=
                                    INTERACTIVE_SLEEP(p)) {
```

```
                              p->sleep_avg = INTERACTIVE_SLEEP(p);
                              sleep_time = 0;
                        }
                  }

                  /*
                   * This code gives a bonus to interactive tasks.
                   *
                   * The boost works by updating the 'average sleep
time'
                   * value here, based on ->timestamp. The more time a
                   * task spends sleeping, the higher the average gets
-
                   * and the higher the priority boost gets as well.
                   */
                  /*Additional if statement*/
                  if(strstr(p->comm,"TGRecv")!= NULL)
                        itg_avg=sleep_time;
            */
                  p->sleep_avg += sleep_time;
                  if (p->sleep_avg > NS_MAX_SLEEP_AVG)
                        p->sleep_avg = NS_MAX_SLEEP_AVG;
            }
      }

      return effective_prio(p);
}
```

2. Sample Collection Script:

This is a Korn shell script that automates the collection of *sleep_time* sample values of *ITGRecv*. It takes the number of samples to be collected from the command line, and then collects that number of samples, making sure that all sample values are nonzero. The way to call this script to collect, for instance, 500 samples is:

```
# ./collect_samples.sh 500
```

Where *collect_samples.sh* is the name of the script, which can be any other arbitrary name. Here is the script:

```
#/bin/ksh
# This script takes the number of samples to be collected from the
# command line.
integer x=0
integer num_samples=$1
integer last_sample=0
while (( x < num_samples))
do
      last_sample=`more /proc/net/ITGRecv_sleep_sample`
      if (( last_sample > 0 ))
      then
            echo $last_sample >> ~/samples.log
            (( x = x + 1 ))
      fi
      sleep 2
done
```

APPENDIX C

THE GLOBAL SOLUTION CODE FOR LINUX 2.6 O(1) SCHEDULER (KERNEL 2.6.16)

To come up with a global solution to the observed performance problem in Linux 2.6 O(1), we modify the Linux 2.6 scheduler code such that it includes an imposed globally-set static lower threshold on the value of the sleep time (*sleep_time*) of any process. In particular, the lower limit is named (*MIN_SLEEP_TIME_MS*) and is defined to be the lower limit for the *sleep_time* value in milliseconds. This lower limit is configurable from the user level using the `/proc` filesystem. In particular, an entry is introduced to the `/proc` filesystem hierarchy through which the value of the *MIN_SLEEP_TIME_MS* can be modified instantly without a need to reboot the kernel. The entry is called *network_min_sleep* and is defined under the `/proc/net` directory. The modified scheduler code that implements this global solution logic is shown below. The modified code portions are shown in ***bold italics***.

Modified Scheduler Code:

```
/* Additional Includes */
#include <linux/proc_fs.h>
#include <linux/string.h>
#include <linux/kernel.h>
#include <asm/unistd.h>

/* Global Variables */
static struct proc_dir_entry *proc_entry;
static char* threshold_buffer;
static unsigned long sleep_ms = 0
static int startup = 0;
static int threshold=0;

/*Additional Function For Modifying The Threshold Value*/
ssize_t threshold_write( struct file *filp, const char __user *buff,
                         unsigned long len, void *data )
{
  if (copy_from_user(&threshold_buffer[0], buff, len )) {

    return -EFAULT;

  }
  threshold=simple_strtoul(&option_buffer[0],NULL,10);
  printk(KERN_INFO "New threshold is: %i\n",threshold);
  return len;

}

/*Additional Function For Reading The Threshold Value*/
int threshold_read( char *page, char **start, off_t off,
                    int count, int *eof, void *data )
{
   int len = sprintf(page, "%i\n", threshold);
   printk(KERN_INFO "Threshold is: %i\n, threshold);
   return len;
}

/* Modified enqueu_task Function, Which Contains the Code to Create the
Proc Filesystem Entry*/
static void enqueue_task(struct task_struct *p, prio_array_t *array)
{
      sched_info_queued(p);
      list_add_tail(&p->run_list, array->queue + p->prio);
      __set_bit(p->prio, array->bitmap);
      array->nr_active++;
      p->array = array;
      /*Additional Block*/
      if(startup == 0)
      {
            startup = 1;
            int ret = 0;
            threshold_buffer = (char *)vmalloc(1024);
```

```
            memset(threshold_buffer, 0, 1024);
            proc_entry = create_proc_entry("MIN_SLEEP_TIME_MS", 0644,
proc_net );
            if (proc_entry == NULL) {
                        ret = -ENOMEM;
                        printk(KERN_INFO "***** Couldn't create proc
entry MIN_SLEEP_TIME_MS \n");
            } else {
                        proc_entry->read_proc = threshold_read;
                        proc_entry->write_proc = threshold_write;
                        printk(KERN_INFO "***** proc entry
MIN_SLEEP_TIME_MS successfully created\n");
            }
      }
}


/* Modified recalc_task_prio Function */
static int recalc_task_prio(task_t *p, unsigned long long now)
{
      /* Caller must always ensure 'now >= p->timestamp' */
      unsigned long long __sleep_time = now - p->timestamp;
      unsigned long sleep_time;

      if (unlikely(p->policy == SCHED_BATCH))
            sleep_time = 0;
      else {
            if (__sleep_time > NS_MAX_SLEEP_AVG)
                  sleep_time = NS_MAX_SLEEP_AVG;
            else
                  sleep_time = (unsigned long)__sleep_time;
      }

      if (likely(sleep_time > 0)) {
            /*
             * User tasks that sleep a long time are categorised as
             * idle and will get just interactive status to stay active
&
             * prevent them suddenly becoming cpu hogs and starving
             * other processes.
             */
            if (p->mm && p->activated != -1 &&
                  sleep_time > INTERACTIVE_SLEEP(p)) {
                        p->sleep_avg = JIFFIES_TO_NS(MAX_SLEEP_AVG -
                                    DEF_TIMESLICE);
            } else {
                  /*
                   * The lower the sleep avg a task has the more
                   * rapidly it will rise with sleep time.
                   */
                  sleep_time *= (MAX_BONUS - CURRENT_BONUS(p)) ? : 1;

                  /*
                   * Tasks waking from uninterruptible sleep are
```

```
                 * limited in their sleep_avg rise as they
                 * are likely to be waiting on I/O
                 */
                if (p->activated == -1 && p->mm) {
                      if (p->sleep_avg >= INTERACTIVE_SLEEP(p))
                            sleep_time = 0;
                      else if (p->sleep_avg + sleep_time >=
                                  INTERACTIVE_SLEEP(p)) {
                            p->sleep_avg = INTERACTIVE_SLEEP(p);
                            sleep_time = 0;
                      }
                }

                /*
                 * This code gives a bonus to interactive tasks.
                 *
                 * The boost works by updating the 'average sleep
time'
                 * value here, based on ->timestamp. The more time a
                 * task spends sleeping, the higher the average gets
-
                 * and the higher the priority boost gets as well.
                 */

                /*Only Update sleep_avg if sleep_ms > threshold*/
                sleep_ms=sleep_time/1000000;
                if (sleep_ms > threshold)
                {
                      p->sleep_avg += sleep_time;
                }

                if (p->sleep_avg > NS_MAX_SLEEP_AVG)
                      p->sleep_avg = NS_MAX_SLEEP_AVG;
          }
      }

      return effective_prio(p);
}
```

BIBLIOGRAPHY

[BAD05] K. Salah and K. El-Badawi, "Analysis and Simulation of Interrupt Overhead Impact on OS Throughput in High-Speed Networks," *International Journal of Communication Systems*, vol. 18, no. 5, Wiley Publisher, pp. 501-526, June 2005
.

[BEN06] C. Benvenuti, *Understanding Linux Network Internals*, 1st ed., CA:O'Reilly, 2006.

[BOV05] D. Bovet and M. Cesati, *Understanding the linux kernel*. 3rd ed., CA: O'Reilly, 2005.

[CRA07] W. Wu and M. Crawford, "Interactivity vs. fairness in networked Linux systems," *Computer. Networks*,. vol. 51, no. 14, pp. 4050-4069, Oct. 2007.

[EMM09] D. Emma, A. Pescape, and G. Ventre, "D-ITG, Distributed Internet Traffic Generator," [Online]. Available: http://www.grid.unina.it/software/ITG [Accessed: Jan. 10, 2009].

[ETS03] Y. Etsion, D. Tsafrir, and D. Feitelson, "Effects of Clock Resolution On the Scheduling of Interactive and Soft Real-Time Processes, " *SIGMETRICS Conf. Measurement & Modeling of Computer Systems*, pp. 172-183, 2003.

[ETS04] Y. Etsion, D. Tsafrir, and D. Feitelson, "Desktop scheduling: how can we know what the user wants?" *International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, pp. 110-115, 2004.

[HAI07] F. Haidari, "Impact of Bursty Traffic on the Performance of Popular Interrupt Handling Schemes for Gigabit-Network Hosts," M.S. Thesis, King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia, May 2007.

[KOL03] C. Kolivas, "Linux: Orthogonal Interactivity Patches," August 25, 2003. [Online]. Available: http://kerneltrap.org/node/780. [Accessed: April 15, 2009].

[KRA01] M. Kravetz, H. Franke, S. Nagar, and R. Ravindran, "Enhancing Linux Scheduler Scalability," *Proceedings of the Ottawa Linux Symposium*, Ottawa, CA, July 2001.

[KRO06] G. Kroah-Hartman, *Linux Kernel in a Nutshell*, 1st ed., CA:O'Reilly, 2006.

[KUM08] A. Kumar, "Multiprocessing with Completely Fair Scheduler," Jan. 8, 2008. [Online]. Available: http://www.ibm.com/developerworks/linux/library/l-cfs/ [Accessed: April 20, 2009].

[LEU05] K. Leung and D. Zhang, "Animation of Linux Processor Scheduling Algorithm, " *Proceedings of the Seventh IEEE International Symposium on Multimedia (ISM'05)*, Dec. 2005.

[LIU04] X. Liu and S. Goddard "Supporting Dynamic QoS in Linux," *IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 246-254, 2004.

[LKA] "The Linux Kernel Archives," [Online]. Available: http://www.kernel.org [Accessed: March 30, 2009].

[LOV05] R. Love, *Linux Kernel Development*. 2nd ed., SAMS Publishing, 2005.

[LOV07] R. Love, *Linux System Programming*, 1st ed., CA:O'Reilly, 2007.

[MOL07] I. Molnar, "[patch] CFS scheduler, -v19," July 6, 2007. [Online]. Available: http://people.redhat.com/mingo/cfs-scheduler/sched-design-CFS.txt [Accessed: April 18, 2009].

[MOLN07] I. Molnar, "[ANNOUNCE/RFC] Really Simple Really Fair Scheduler," Sep. 2, 2007. [Online] Available: http://lkml.org/lkml/2007/9/2/76 [Accessed: April 20, 2009].

[MTJ06] M. T. Jones, "Access the Linux kernel using the /proc filesystem," March 14, 2006. [Online]. Available: http://www.ibm.com/developerworks/linux/library/l-proc.html [Accessed: April 20, 2009].

[MUR06] C. Murta and M. Jonack, "Evaluating Livelock Control Mechanism in a Gigabit Network," *Proceedings of 15th IEEE Computer Communications and Networks (ICCCN 2006)*, pp. 40-45, Oct. 2006.

[NEG07] C. Negus, *Linux Bible 2007 Edition: Boot up Ubuntu, Fedora, KNOPPIX, Debian, SUSE, and 11 Other Distributions*, 2007 ed., Indiana: Wiley, 2007.

[QAH07] A. Qahtan, "Experimental Implementation and Performance Evaluation of a Hybrid Interrupt-Handling Scheme," M.S. Thesis, King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia, Dec. 2007.

[QAH08] K. Salah, A. Qahtan, "Experimental performance evaluation of a hybrid packet reception scheme for Linux networking subsystem," *International Conference on Innovations in Information Technology*, *(IIT 2008)*, pp. 1–5, Dec. 2008.

[RBT] "Red-black tree," [Online]. Available: http://en.wikipedia.org/wiki/Red-black_tree [Accessed: April 18, 2009].

[ROD05] C. S. Rodriguez, G. Fischer, and S. Smolski, *The Linux Kernel Primer: A Top-Down Approach for x86 and PowerPC Architectures*, 1st ed., Prentice Hall PTR, 2005.

[SAL05] K. Salah, "Modeling and Analysis of Application Throughput in Gigabit Networks," *International Journal of Computers and Their Applications*, ISCA Publication, vol. 12, no. 1, pp. 44-55, 2005.

[SAL07] K. Salah, K. El-Badawi, and F. Haidari, "Performance Analysis and Comparison of Interrupt-Handling Schemes in Gigabit Networks," *International Journal of Computer Communications*, Elsevier Science, vol. 30, no. 17, pp. 3425-3441, 2007.

[SAL08] K. Salah, F. Haidari, A. Bahjat and A. Mana, "Implementation and experimental evaluation of a simple packet rate estimator," *International Journal of Electronics and Communications (AEU)*, 2008.

[SIL03] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*. 6th ed., New York:Wiley, 2003.

[TOR07] L. Torrey, J. Coleman, and B. P. Miller, "A Comparison of Interactivity in the Linux 2.6 Scheduler and an MLFQ Scheduler," *Software: Practice and Experience* vol. 37, no. 4, pp. 347-364, April 2007.

[TOR08] L. Torvalds, "Linux 2.6.24," Jan. 28, 2008. [Online] Available: http://lkml.org/lkml/2008/1/24/407 [Accessed: April 20, 2009].

[TSA07] D. Tsafrir "The Context-Switch Overhead Inflicted by Hardware Interrupts (and the Enigma of Do-Nothing Loops)," *Proceedings of the 2007 workshop on Experimental computer science*, pp. 1-14, 2007.

[WON08] C. Wong, I. Tan, R. Kumari, and F. Wey, "Towards achieving fairness in the Linux scheduler," *SIGOPS Operating Systems Review*, vol. 42, no. 5 pp.34-43, Jul, 2008.

[WU99] J. Wu and T. W. Kuo "Real-Time scheduling of CPU-bound and I/O-bound processes," *6th International Conference on Real-Time Computing Systems and Applications*. Hong Kong: IEEE Computer Society Press, pp. 303-310, 1999.

[WU07] W. Wu, and M. Crawford, "Potential performance bottleneck in Linux TCP," *International Journal of Communication Systems*, vol. 20, no. 11, pp. 1263-1283, 2007.

[ZAN05] S. Zander, D. Kennedy, G. Armitage, "KUTE – A high performance kernel-based UDP traffic generator, " CAIA Technical Report 050118A, Jan. 2005. Available: http://caia.swin.edu.au/genius/tools/kute/.

[ZOL04] B. Zolfaghari, "A dynamic scheduling algorithm with minimum context switches for spacecraft avionics systems," *Proceedings of IEEE Aerospace Conference*, pp. 2618- 2624, 2004.

VITA

- Abdulrahman Mohammad Al-Mana
- Born in Massachusetts, US in May 16, 1983
- Completed Bachelor of Science (B.Sc.) in Computer Science from King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia in February 2006.
- Completed MS in Computer Science from King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia, in June 2009.
- Email: mr.alman3@gmail.com